(4)

Ada 9X Project Report

Ada 9X Project
Revision Request Report

Supplement 1

January 1990

Office of the Under Secretary of Defense for Acquisition

Washington, D. C. 20301

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1990	Final 25 Jul to 31 Oct 1989

4. TITLE AND SUBTITLE

Ada 9X Project Revision Request Report
Supplement 1

5. FUNDING NUMBERS

C = MDA-903-87D-0056

6. AUTHOR(S)

Compiled by IIT Research Institute

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
1211 South Fern St., 3E113
The Pentagon
Washington, DC 20301-3080

Ada 9X Project Office
AF Armament Lab/FXG
Eglin AFB, Florida 32542-5434

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

This report is a supplement to the August 1989 report with the same title.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document contains the revision requests received by the Ada 9X Project Office between 25 July and 31 October 1989 for consideration under the current review of ANSI/MIL-STD-1815A. Revision requests received between 20 October and 24 July 1989 are published in an initial report, Ada 9X Project Revision Request Report, August 1989.

14. SUBJECT TERMS

Ada 9X Project, Revision Requests, ANSI/MIL-STD-1815A, Ada Joint Program Office, Lexical Elements, Declaration and Types, Names and Expressions, Statements, Subprograms, Packages, Visibility Rules, Tasks, Program Structure and Compilation Issues, Exceptions, Generic Units, Representation Clauses and Implementation-Dependent Features, Input-Output

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UL	

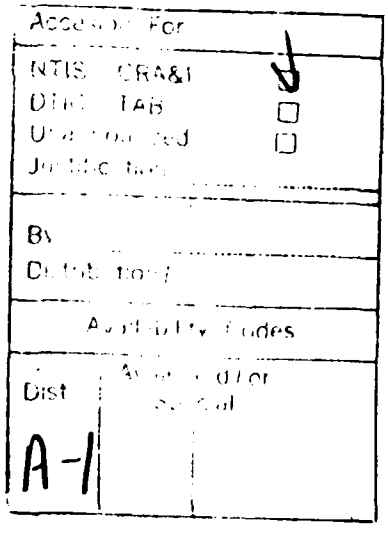
PREFACE

This document contains the revision requests received by the Ada 9X Project Office between 25 July and 31 October 1989 for consideration under the current review of ANSI/MIL-STD-1815A. Revision requests received between 20 October 1988 and 24 July 1989 are published in an initial report, Ada 9X Project: Revision Request Report, August 1989.

To obtain copies of this report, or other Ada 9X Project reports, contact the Ada Information Clearinghouse at 703-685-1477. The revision requests, additional information on their current status, and status reports of the Ada 9X Project's achievements, are available on the Ada 9X electronic bulletin board at 1-800-Ada9X25 or 1-301-459-8939, on the AJPO host on Internet, or on Eurokom.

This document was prepared by IIT Research Institute (IITRI) under sponsorship of the Ada 9X Project Office. The IITRI Program Manager is Mary Armstrong. IITRI staff that contributed to this report are Susan Carlson, Trisha Guethlein, and Hank Greene.

The Ada 9X Project Office is directed by Chris Anderson at the Air Force Armament Laboratory, Eglin Air Force Base, Florida 32542-5434. The Ada 9X Project is sponsored by the Ada Joint Program Office.



INTRODUCTION

BACKGROUND

The Ada 9X Project was initiated in October 1988 to revise ANSI/MIL-STD-1815A. The Ada9X Project Office was established at the Air Force Armament Laboratory under the direction of Chris Anderson. The overall goal of the Ada 9X Project is to revise ANSI/MIL-STD-1815A to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community. The Ada 9X process is a revision and not a redesign of the language and should be viewed as a natural part of the maturation process.

Requirements for the revision are based on input from the Ada community in the form of special studies, workshops, public meetings, Ada Language Issues, and revision requests. The Ada 9X Project Office conducted an "open call" for revision requests from October 1988 to October 1989. Individuals and groups were encouraged to submit requests for a particular revision of the language using the format shown in Appendix A. Revision requests are currently being reviewed by the Ada 9X Project Requirements Team. The status of revision requests will be tracked throughout the Ada 9X revision process. Revision requests may be viewed on the Ada 9X Project electronic bulletin board at 1-800-Ada9X25 or 1-301-459-8939, on the AJPO host on Internet, or on Eurokom.

ORGANIZATION OF THIS DOCUMENT

This document contains Revision Requests 0151 through 0837. Requests are organized according to relevant sections in the Language Reference Manual, ANSI/MIL-STD-1815A. If a request is relevant to more than one section, it will physically appear in the first section designated by the author and appear by reference in other sections. Indices are provided to support reference by key technical terms, revision request number, revision request title, submitter, and organization. Instructions for incorporating this supplement into the Ada 9X Project Revision Request Report, August 1989 are given on page vi. Changes to the August 1989 report are provided as Attachment I.

INSTRUCTIONS FOR INCORPORATING THIS SUPPLEMENT REPORT INTO ADA9X PROJECT REVISION REQUEST REPORT, AUGUST 1989

REMOVE	INSERT
i to vii	i to viii
1-6	1-6 to 1-26
2-6 to 2-8	2-6 to 2-33
3-46 to 3-48	3-46 to 3-269
4-26	4-26 to 4-133
5-10	5-10 to 5-54
6-22 to 6-23	6-22 to 6-118
7-14	7-14 to 7-77
8-10	8-10 to 8-64
9-56	9-56 to 9-161
10-20	10-20 to 10-79
11-16 to 11-18	11-16 to 11-55
12-4 to 12-6	12-4 to 12-67
13-28 to 13-29	13-28 to 13-95
14-8	14-8 to 14-54
-	15-12 to 15-38
16-18	16-18 to 16-133
-	17-1 to 17-150
I-1 to I-30	I-1 to I-124

TABLE OF CONTENTS

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 1. INTRODUCTION

while not being erroneous. Hence an implementation could still use either mechanism, but the semantics would be either copy or reference (for any specific call). The freedom given by the current language is far more than implementors require. For instance, as it stands, an implementation could choose the mechanism dynamically, even if the subprogram was compiled in-line.

IMPORTANCE: IMPORTANT

But ESSENTIAL for safety-critical software.

CURRENT WORKAROUNDS:

In the long-term, program analysis tools could make some impact on the problem. No effort is known by the author to detect erroneous programs for the full language. Hence there are not effective workarounds known.

POSSIBLE SOLUTIONS:

It is felt that a significant improvement over the current position could be attained by requiring implementations to define critical properties related to this area. For instance, the implementation could be required to define the parameter mechanism used, or define the order of elaboration of library units.

ISO WG9 be asked to study the implications of erroneous execution and incorrect order dependence upon the effective use of Ada.

ISO WG9 be asked to include in its agreed work item on the uniformity of implementations to prepare proposals for Ada 9X to reduce the impact of erroneous programs and those with incorrect order dependence.

TRUE TYPE RENAMING

DATE: August 2, 1989

NAME: James W. McKelvey

ADDRESS: R & D Associates
P.O. Box 5158
Pasadena, CA 91107

TELEPHONE: (818) 397-7246

ANSI/MIL-STD-1815A REFERENCE: 1, 8.5(9), 8.5(16), 2, 6.4.1(3), 6.3.1(3)

PROBLEM:

Renaming of types using the subtype declaration method (8.5, 16) is not a true renaming. There are at least two problem areas:

1. When "renaming" an enumerated type using the subtype declaration method, the enumeration literals are not automatically renamed as well. While these can be renamed as functions, the result requires far too much effort than is reasonable. Furthermore, the renamed functions are not equivalent to the original literals in that they may not be used in CASE statements.

2. A "renamed" type is not acceptable in an actual parameter type conversion because the semantics require the strong condition of conformance ((6.3.1, 3), the type is required, a subtype of it will not do). A true renaming would eliminate this surprising limitation.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

1. Enumeration literals may be renamed as functions. Alternatively, constants of the "renamed" subtype may be declared and assigned values from the original type. The latter solution is required when the literals are to be used in CASE statements.

2. Actual parameter type conversions cannot use "renamed" types; the original type must be used. This usually requires expanded names, the avoidance of which motivated the "renaming" in the first place.

POSSIBLE SOLUTIONS:

Allow "renames" to be used with subtypes. A statement like:

```
subtype X_Type renames The_Package.X_Type;
```

would make a "new name" for the The_Package.X_Type, and import its enumeration literals, if any. This new name could substitute for The_Package.X_Type in any context.

A statement like:

```
type X_Type renames The_Package.X_Type;
```

would remain illegal.

ADA LINE OF CODE (ALOC) STANDARD

DATE: October 12, 1989

NAME: Craig Cowden

ADDRESS: Naval Security Group Detachment
Pensacola, Florida 32511

TELEPHONE: (904) 452-6399

ANSI/MIL-STD-1815A REFERENCE: 1

PROBLEM:

Many software engineering principles and management decisions are based on the vague concept of a line of Code (LOC). There currently is a proliferation of methods within the Ada community to measure the size of Ada projects. A few examples of the definition of a LOC include:

- Lines (A simple count of carriage returns.)

- Statements (Lines minus comment lines.)

- Source Instructions (A count of terminating semicolons.)

- Delivered Source Instructions (Lines containing actual code.)

The lack of a common standard definition makes function such as life-cycle support, development cost estimation, and product evaluation more difficult.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

When evaluating Ada software, an organization can normalize the LOC size to some set standard using a coefficient. This assumes:

- The method of measurement is known.

- There is enough statistical data to correlate the measurement method with the organization standard.

POSSIBLE SOLUTIONS:

Include a precise definition of an "Ada Lines of Code" in ANSI/MIL-STD-1815A (The Ada language Reference Manual). For example, Section 1.7 could be added to Chapter 1 to read:

1.7 Ada Source Line of Code (ASLOC)

The language defines the total number of lines in an Ada program as the Ada Lines of Code (ALOC).

This measure is a simple count of end of line characters and includes code lines, comment lines and blank lines.

An Ada Source Line of Code (ASLOC) is defined as a simple or compound statement, an exception handler, a data declaration, or an Ada component declaration such as a package, task, generic, or subprogram. This measure can be computed by counting terminating semicolons (";").

DIAGNOSIS OF INCORRECT SYNTAX OR SEMANTICS

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 1.1.2

PROBLEM:

Some compilers identify errors with catch-all messages such as "syntax error", or fail to display data associated with the error, e.g., the name of an undeclared object. Some compilers generate large numbers of spurious error messages after detecting an error, e.g., after detecting an undefined variable in the expression of an if statement.

IMPORTANCE: IMPORTANT

Poor compilers make Ada harder to learn, and reduce the productivity of all programmers, even the more experienced.

CURRENT WORKAROUNDS:

Shop around when looking for an Ada compiler. Unfortunately, there are very few vendors.

POSSIBLE SOLUTIONS:

Add to 1.1.2 a list of errors that a conforming compiler is required to diagnose. Prescribe the minimum acceptable level of detail that must be written for each of those errors. Add a list of errors for which the compiler must have adequate recovery. As part of the Ada Compiler Validation Suite, test and report on the quality of error diagnosis. Indicate that vendors are encouraged to diagnose errors more precisely than required by the standard.

ALLOW CONTROLLED SUBSETS AND SUPERSETS

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 1.1.2

PROBLEM:

The current standard prohibits subsets and supersets, even for a compiler that supports the full language and is capable of enforcing compliance with the standard.

IMPORTANCE: IMPORTANT

Prohibiting supersets discourages experiments with new language constructs. As a result, the development of Ada revision, e.g., Ada 200x, must occur without the benefit of experience, and more prototyping will be necessary in order to evaluate proposed language extensions. Further, there is less incentive to think about enhancements to the language if they cannot be commercially exploited.

Prohibiting subsets makes it harder to enforce local coding standards. If an installation has no other way to prevent use of Ada features that are expensive in a particular environment, it may resolve the problem by selecting a different implementation language.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Change 1.1.2(1) to read

A conforming implementation is one that, when run with default options: ....

DIAGNOSIS OF QUESTIONABLE SYNTAX OR SEMANTICS

DATE: August 31, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4/64
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 1.1.2

PROBLEM:

Some programs contain constructs that are legal but potentially dangerous, or that are likely to be inadvertant, e.g., loss of visibility to identifiers in STANDARD.

IMPORTANCE: IMPORTANT

Without a means for flagging such situations, errors will be harder to detect and software development in Ada will be more costly and less reliable.

CURRENT WORKAROUNDS:

Shop around when looking for an Ada compiler. Unfortunately, there are very few vendors.

POSSIBLE SOLUTIONS:

Add to 1.1.2 a list of questionable usages that a conforming compiler is required to diagnose, when requested by a compiler option. Prescribe the minimum acceptable level of detail that must be written for each of those usages. Prescribe the minimum granularity for requesting this diagnosis. As part of the Ada Compiler Validation Suite, test and report on the quality of such diagnosis. Indicate that vendors are encouraged to diagnose questionable usages more completely than required by the standard.

SUBSETS RECOMMENDED

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 1.1.2

PROBLEM:

The size of an Ada compiler, library manager and runtime system are very large and, therefore, very expensive to develop. This reduces the number of compiler vendors that compete for a given hardware platform. Some hardware will never have an Ada compiler due to the size and price constraints.

IMPORTANCE: ADMINISTRATIVE

This problem most directly affects the cost of compilation systems. Somewhat indirectly, the cost of learning Ada is Also affected.

CURRENT WORKAROUNDS: NONE

(According to the current standard there shouldn't be any, although the "no Chapter 13" subset is quite common.)

POSSIBLE SOLUTIONS:

Allow one (or several) subset(s) of Ada which could be implemented and validated separately. Make each level a proper subset of the next level. Suggested levels:

Full Ada: The fully implemented language including the chapter 13 features.

Level 1: The currently required level of implementation. (not requiring chapter 13 features.)

Level 2: Level 1 without tasking.

Level 3: Level 2 without generics but requiring (pseudo) instantiation of the predefined library generics.

EXTENSIONS WHEN LENGTH CLAUSES ARE USED

DATE: August 10, 1989

NAME: E B Pitty

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833483
E-mail: epitty@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 1.2

PROBLEM:

Currently compilers are not required to take into account the effects of sign extension when length clauses are used. For example, it is desired to associate an Ada enumerated type with various condition codes in a 32-bit host architecture. The literals of three valued enumerated type are specified as mapping to the values -1, 0, 1. If a length clause is then used to specify objects of this type are to occupy 32 bits, this may not be sufficient to ensure values can be correctly passed between the program and the host Specifically, the compiler may (legitimately) use only the lowest say 8-bits to represent the three values, (i.e. "11111111", "00000000", "10000000"), with the remaining 24-bits be all zero. Consequently the host will see the values 255,0,1 as opposed to 1,0,1.

IMPORTANCE: IMPORTANT

Limitations in the use of Ada with embedded systems or in interaction with other programming languages.

CURRENT WORKAROUNDS:

Can be avoided by explicitly 'coding' mapping between types as opposed to using length/representation clauses.

POSSIBLE SOLUTIONS:

For the scenario given above it would be sufficient for the compiler to fully use the 32-bits by sign extending any negative numbers. However this is dependent upon the compiler and host sharing a common representation for integers, e.g. 2's-complement.

ATOMIC TRANSACTIONS

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 1.3

PROBLEM:

Mutually exclusive control is often necessary in embedded systems with applications sharing data, devices, IO buffers, processes, etc. Ada has no way to easily support cases where highly efficient, mutually exclusive activities are necessary. Often, the hardware contains special instructions, such as, Test & Set, but the Ada grammar does not allow the programmer to write a recognizable construct for the compiler to know to produce the desired mutually exclusive protection. The problem is even worse for separate compilation units where the code generator does not have visibility.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The only approach in Ada is to add much code to synchronize a mutually exclusive process by way of a rendezvous for a protected access. This approach draws in all of the runtime support and overhead for a tasking rendezvous without guaranteeing that you will not have deadlocks or other task control management problems.

POSSIBLE SOLUTION:

Allow the addition of a new term "atomic" which will signal the mutually exclusive control and access of a particular object.

MORE FLEXIBLE NOTATION FOR SYNTAX

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 1.5

PROBLEM:

The notation used for defining Ada syntax does not allow a clean way of handling certain common situations, e.g., alternatives within a larger expression.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Define additional syntactic categories.

POSSIBLE SOLUTIONS:

Specify syntactic bracketing characters, similar to the <> brackets of BNF, e.g., allow something like

```
A ::= w <x|y> z
```

Specify a permutation notation for a sequence of syntactic categories that may occur in any order.

Provide a means to specify that a particular syntactic category not appear.

Use an attribute grammar. This solves a lot of problems, but it is a rather drastic change.

REQUIRED REPORTING OF CERTAIN EXCEPTIONS

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 1.6(s)

PROBLEM:

A compiler is not required to report a known, certain-to-be-raised exception, according to the last sentence in 1.6(s).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Require compilers to report (or have capability to report) these "certain" exceptions. Knowing this at compile-time will save debug time and make the final code more maintainable.

STATIC SEMANTICS AND SUPPORT FOR FORMAL ANALYSIS

DATE: July 24, 1989

NAME: D J Tombs, and endorsed by Ada-UK

ADDRESS: RSRE
ST. Andrews Road
Great Malvern
Worcestershire
WR14 3PS, UK

TELEPHONE: +44 684 895311

ANSI/MIL-STD-1815A REFERENCE: 1.6, 8 (but not specifically)

PROBLEM:

If Ada is to be used in safety-critical systems then there must be formal methods which are applicable to Ada programs (as requested by UK Defence Standard 00-55 [2]). An example would be a tool to detect any uses of recursion or dynamic storage, which are undesirable in a safety-critical program but tedious and difficult to detect by hand. In addition, there are many applications outside the safety-critical domain where some higher degree of program surety than mere human inspection is desired. Yet Ada is so large and complex that it is difficult to give a reasonable formal specification for the static semantics of even small parts of the language. Program analysis and verification need such a specification.

There are two main sources of problem:

(i) where the informal (LRM) definition is long and difficult, but where the meaning of the program is sorted out by a compiler front end. Here we are thinking of problems of name visibility and overloading (LRM sect.8).

(ii) where the semantic behavior is deliberately left undefined. Here we are thinking of the profusion of erroneous or order-dependent constructs in Ada, and the differing implementations allowed by LRM sect. 1.6.[3,6]

IMPORTANCE: ESSENTIAL

This request is essential if it is intended that Ada9X should be capable of at least some degree of formal specification and analysis.

CURRENT WORKAROUNDS:

To date, the author is unaware of any formal toolsets which act on the full Ada language. Most analysis has been done using a formally specified intermediate language, such as MALPAS IL Under this approach Ada is compiled to the intermediate language, but with an unspecified mapping, proofs can then only apply to the intermediate language, although there may be some means of relating the intermediate code and the source Ada.

Another approach is to take a small, well-specified language with an Ada-like syntax in which legal programs

can be compiled with a validated Ada compiler. This is the attitude taken by SPARK [4] and AVA [5]. For this approach to work the small language must avoid the Ada constructs which introduce the problems noted above, otherwise there can be no guarantee that the full compiler will interpret a source program in a manner consistent with the definition of the small language. Also, a small language might only be used for writing small programs in the safety-critical field, where it is doubtful that a full validated compiler could be produced which is sufficiently reliable. A small, dedicated compiler might be a better bet.

The ultimate workaround is to use another, more easily defined language.

POSSIBLE SOLUTIONS:

Specifying the overloading and visibility rules is unavoidably difficult. Possibly the best solution to the problem is to have an intermediate language which is Ada-like, but where all overloading and hiding has been removed and where expressions in multiple declarations, subprogram defaults, etc., are expanded out. The mapping from Ada to this intermediate language will be well-defined. A defined expansion of full Ada is part of Dr. Wichmann's Low Ada proposal [6].

The above discussion seems to have little bearing on the Ada9X process, but it would be nice to have some "official" support for attempts to solve the problem.

Conversely, it is part of the Ada9X process to obtain a better definition of what happens when erroneous or order-dependent code is compiled and run. Possible options include:

- specifying the language fully wherever practicable (eg stating the order of evaluation for some expressions or insisting that parameter passing be by reference);
- allowing the programmer to state what is desired to happen by means of pragmas;
- insisting that the compiler be self-consistent (presently an implementation is allowed at run-time to make a random decision on elaboration order!);
- insisting that a validated compiler declare its strategy for at least the majority of erroneous constructs.

This last seems to be the most flexible and the most immediately achievable. Also, it does not preclude optimization for parallel and vector architectures, as might happen under a more rigid regime. Obviously there is much detail to be worked through but it does seem to offer hope of a better situation than presently exists.

Some consideration might also be given to having an "official" dialect of Ada especially for safety-critical purposes, which is formally specified and where undesirable constructs like recursion and dynamic storage are excluded from the definition.

REFERENCES:

[1] Ada9X Revision Request no 89 04 19 0066,, our ref Ada-UK/002
[2] UK Ministry of Defence Interim Defence Standard 00-55 "Requirements for the Procurement of Safety Critical Software in Defence Equipment"
[3] "Catalogue of Ada Runtime Implementation Dependencies" ACM SIGAda Ada Runtime Environment Working Group, December 87
[4] "SPARK - The Spade Ada Kernel" B A Carre, T J Jennings, University of Southampton, UK, March 88
[5] "The nanoAVA Definition" Dan Craigen, Mark Saaltink, Michael K Smith, Computational logic, Austin, Texas, June 88

[6] "Insecurities in the Ada Programming Language", and private Communication, B A Wichmann, National Physical Laboratory, London, August 88

ERROR CLASSIFICATION

DATE: June 9, 1989

NAME: Judy A. Edwards

ADDRESS: General Dynamics
P.O. Box 748, MZ 1746
Fort Worth, TX 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 1.6, 3.6.1 #4, 3.8.1, & 4.5.7

PROBLEM:

Embedded avionics software requires higher degrees of integrity. An error classification in a language definition that allows compiler system's recognizable errors to be propagated into the final linked version without generating warnings is not in keeping with the spirit of more reliable systems. Currently, a compiler has a minimal requirement for early recognition of problems. Then, the user may not discover an inadvertent error until very late in the development cycle when the correction cost is much greater.

The language definition should add a requirement in para. 1.6 to generate a "user warning" at the first moment that a construct may be considered erroneous at compile time, e.g., incompletely specified, incorrect constructs, or exceptions. The compilers should also be robust and attempt to produce code even in light of errors, but it should not deceive the user that all constructs in the module are correct.

Further, most of the ACVC tests only look for exceptions to be raised for such compile time erroneous conditions. In those cases, the compiler should recognize that the only code generation required would be the exception handler portion and/or an error message.

IMPORTANCE: IMPORTANT

Without the information and warning messages, the user may have to go through several intermediate testing situations of the executable code for unexpected transfers to exception handlers to be assured that no surprises will occur in the integrated product.

CURRENT WORKAROUNDS:

Compile and test under the VAX hosted compiler, then retest with the cross compiler with an emulated target. Finally, retest on the target to see if any breakpoints on exceptions can be reached. Provide programming standards to trap those classes of error to special exception handlers. Also, the absence of this capability will need special debug tools for tracking exceptions raised such as program_error, constraint_error, elaboration_error, and tasking_error.

POSSIBLE SOLUTIONS:

<<minor impact to the language>>

1. Expand the classes of errors to include warning messages to users to be initiated at the earliest point that the compilation system (including the linker) recognizes incorrect, or potentially incorrect, units.

<<moderate impact>>

2. Null ranges are a major source for constraint and program errors. Therefore, do not allow programmers to write "Null" ranges without a new construct, e.g., NULL. Flag the error/ warning at the earliest level that the compilation system can determine a problem.

3. Delete error-prone constructs and semantics from the language, e.g., places where elaboration errors can occur.

COMPILE-TIME DETECTION OF CONSTRAINT ERRORS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 1.6(4), 3.3.2(6..9)

PROBLEM:

Many constraint errors are detectable at compile time but it is currently acceptable under the standard to defer their detection until run time, when it is potentially very hazardous for the error to occur.

IMPORTANCE: ESSENTIAL

It makes no sense for the Ada language to be so carefully designed for reliability, robustness, and embedded applications (such as satellites, which are notoriously difficult to hook a debugging terminal to once deployed) and then allow the following to compile:

```
Foo : Positive := -1;
```

CURRENT WORKAROUNDS:

Some compiler issue warnings for inevitable constraint errors that are detected at compile time. Some tools are available that provide additional analysis of such errors (much as "lint" provides additionally analysis of errors in C programs); Unfortunately, the effectiveness of such tools depends on the maturity of the programming community's attitude toward their use.

POSSIBLE SOLUTIONS:

eword LRM 1.6.3(b) and 3.3.2(6..9) to indicate that compilers must perform reasonable analysis of potential constraint errors at compile time.[1]

---

[1]Yes, I admit that the term "reasonable" is subjective and (therefore worthless in a standard), but you said this section was optional!

Actually, it might be possible to say that compilers "Shall detect all inevitable constraint errors that are detectable by static analysis." This would permit constraint errors that are only detectable dynamically (at run time) to slip by, but provide a high degree of confidence about dumb errors such as the example given earlier.

COMPATIBILITY:

The proposed solution is quasi-compatible. Some previously-compiled code will re-compile successfully some will not: that which does not was inherently dangerous and destined to fail, so the non-compatibility should be welcomed (as it increases robustness). Execution behavior will also be affected, since errors that used to occur during run-time will now occur during compile time: these changes should also be seen as desirable.

For additional references to Section 1. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 2. LEXICAL ELEMENTS

```
argument_association ::= [ argument_identifier =>] name |
                         [ argument_identifier =>] expression.
```

We propose the FORTRAN pragma

```
pragma FORTRAN (NAMEF [,( A1 [ ,( B1, aggregate { , BM, aggregate})]
                        {,AR [ ,(BR1, aggregate { ,BRM, aggregate})]}
                       )
                      ]
               )
```

where A1, ..., AR correspond to the dummy arguments of NAMEF, and B1, ..., BM, and BR1,...BRM correspond to arrays of a dummy subprogram, with bounds given in the subsequent aggregate (for each dimension lower and upper bounds must be given, all separated by commas), of the formal generic subprograms. NAMEF denotes the name of the coupled FORTRAN subprogram.

Name restrictions are imposed by those of FORTRAN. The FORTRAN pragma is an extension of the INTERFACE pragma and aimed at generic subprograms with only procedures or functions as generic formal parameters; as extension of the INTERFACE pragma it may be used for the cases treated in section 3, by substitution of

pragma INTERFACE(FORTRAN, NAME) by **pragma** FORTRAN(NAME),

It is a pity that in Ada subprograms are allowed as parameters of subprograms only via generic units. A large class of numerical problems such as : quadrature, zero finding, optimization, and solving differential and integral equations, parameterize over a function. Routines in FORTRAN for the above problems parameterize in general via subprograms as dummy arguments. When a dummy subprogram itself contains an array as dummy argument, one must supply information about the bounds, because FORTRAN generally lacks this information. (This is done via the above mentioned aggregates.)

It must be kept in mind that use of the FORTRAN pragma with generic units severely restricts the generic formal parameters: together with the FORTRAN pragma only functions and subprograms are allowed as generic formal parameters, because that is what we need in order to couple FORTRAN subprograms with dummy subprograms and it will probably relieve the implementation of the FORTRAN pragma.

DECOUPLE ADA FROM CHARACTER SET

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 2.1, 2.5, 3.5.2

PROBLEM:

The current LRM restricts Ada to the ASCII 7-bit character set. This makes use of Ada awkward on machines with other character sets, e.g., EBCDIC, and will automatically render the Ada LRM obsolete every time ANSI or ISO revises ASCII/ISCII.

IMPORTANCE: IMPORTANT

Without this capability, users of machines with EBCDIC, 12-bit or 16-bit characters will tend to favor other languages over Ada. Where use of Ada is mandated, reliability will suffer due to the additional code needed in order to deal with private character types. Adherence to national and international standards for character sets will also suffer.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Revise the standard to specify that STANDARD.character is an enumeration type that includes all the current ASCII characters, and to specify the sequence of the first 128 enumeration literals, without precluding additional characters. Specify the characters that a conforming implementation must allow in literals and in constants, without proscribing additional graphic characters. Specify any restrictions on national alphabetic characters. Allow all format effectors where ASCII format effectors are allowed.

ALLOW NATIONAL CHARACTERS IN LITERALS, COMMENTS AND IDENTIFIERS

DATE: March 25, 1989

NAME: Erland Sommarskog

ADDRESS: ENEA Data AB
Box 232
S-183 23 T[BY ("[" = "A" with dots.)
SWEDEN

TELEPHONE: +46-8-7922500 (from April 22th, 1989)
E-mail: sommar@enea.se

ANSI/MIL-STD-1815A REFERENCE: 2.1, 2.3, 2.5, 2.6

PROBLEM:

Today Ada disallows non-printing characters in string literals and comments. As "non-printing" are defined all characters outside the range 32-126 in ASCII. This is very unfortunate since many environments provide printing characters outside this range. Particularly they are used for characters that are common in other languages like English. (E.g. "A" with dots in German, Finnish and Swedish, "E" with accents in French.) Thus, they are very likely to be used in programs which manipulates texts in these languages. However Ada does not permit them, even in string constants, which makes use of them very troublesome.

Furthermore, these characters are often placed in the range 128-255, which is outside the range of the predefined type Character.

Most of these eight-bit sets are defined as ASCII in the bottom half from 0 to 127, and special characters in the upper half. Most important of these are the nine sets defined by ISO 8859. But the language should cause as few obstacles as possible for any character set.

One could divide the problems into three areas: Literals, comments and identifiers. In the case of literals, as already has been said, application requirements are the reason to use an extended set.

As for comments and identifiers, being able to use the full range of the (non-English) alphabet in names and comments makes the program much easier to read.

A problem which is beyond my scope, but must be remembered is the size of character variables. This is normally eight bits, which is too little for languages like Japanese and Chinese. A minimum requirement for these languages must be the possibility to use their characters in strings and comments. I know too little to say anything about their occurrence for identifiers.

IMPORTANCE: ESSENTIAL/IMPORTANT

String literals and comments: ESSENTIAL. The situation today is totally unsatisfactory and if any change should be made to the standard right today, it is this one.

Identifiers: (Very) IMPORTANT. I don't rank this as "essential", since I know there might be conflicting

requirements. Particularly since I know there is a risk that a program that uses national characters is less portable. I still feel that this problem is less than not being allowed to use my native language in entirety in my Ada code.

CURRENT WORKAROUNDS:

Comments and identifiers: Use English. Often desired of other reasons, for instance if code and documentation is to be read by persons from other countries. Yet, in many cases unsatisfactory. The programmer is often much more fluent in his own language than English. And the English produced may not always be understandable by a native English speaker.

Another workaround is to drop accents, dots etc. This gives a poor language which also is harder to read.

String literals: Define an extended character type with character literals for the ASCII characters and enumeration types for the others. Define a new string type to incorporate the new "characters". Write a new Text_io to handle I/0 of the new "characters". This solution is totally unsatisfactory for such a common problem. Being forced to write a string as

"H" & LC_E_ACUTE & "1" & LC_E_GRAVE" & "ne"

is tedious, error-prone and hard to read.

A simpler solution is define character constants with Unchecked_ conversion from the integer codes and then pray that the compiler is not too keen on doing range checks on character variables.

This is of course not very portable.

POSSIBLE SOLUTIONS:

1) Extend the predefined type Character to have 255 elements. This will break programs that rely on that character have 128 elements, for instance a program declares an

   ARRAY (character) OF Some_type

   and then is sensitive for the size of the array. Hopefully, this is a rare problem.

   An alternative as providing a new type, say Character_8, is hardly satisfactory, since would indicate that an eight-bit character has something special, when it should be the normal case.

2) (Solution taken away.)

3) Remove the restriction that only printable character are allowed in strings. Any character should be allowed except the line terminator. It is on the user's response that the program is portable in this sense or not.

4) Whenever possible Ada should regard an eight-bit character code as just that. It is beyond the scope of Ada to say that if the code 65 is output an "A" is being displayed. The only case Ada must put meaning into the codes is when interpreting of the program code itself (except string literals and comments). In this case Ada should rely on ISO 8859 instead of ASCII today.

5) Provide predefined packages like the current ASCII for the nine character sets defined by ISO

8859. (All of them have ASCII in positions 0-127, control characters in 128-159, then various characters in the positions 160-255.

6) Extend the set of characters allowed in identifiers based on ISO 8859. This is where things are getting tricky. If Ada like C and Modula-2 had been case-sensitive, the problems had easily been solved by allowing all characters in the range 161-255.

Of the nine sets in ISO 8859, Latin-1 will be the most commonly used. In this set the characters 192-214 and 216-222 are paired with a lower-case correspondent 32 positions up. 223 (German double S) and 255 ("y" with diaeresis) are unpaired. 215 and 247 are non-letters.

The situation is the same in Latin-2, 3 and 4, except that there are also letters in the range 160-191. In this case uppercase and lowercase are paired with a distance of 16. I don't know about the Cyrillian, Arabic, Hebrew, Greek and Latin-5 sets. One solution to this dilemma is to have a pragma to tell which character set is in use. The set would then define which characters were allowed. It is easy to see that this solution is not workable. Packages not based on the same character set would get difficulties to cooperate.

Restrict Ada to 8859/1 (Latin-1) only. This makes it simple for Ada, but is unfriendly to people in Eastern Europe, Russia, Greece, Arabia and Israel.

Extract the rules from the standard if possible. As far as Latin-1 to 4 it is, although it would allow for strange names in Latin-1. But, once again, that is on the programmer's response, not on the language.

NATIONAL LANGUAGE CHARACTER SETS

DATE: September 1989

NAME: Randal Leavitt (Canadian AWG #003)

ADDRESS: PRIOR Data Sciences Ltd.
240 Michael Cowpland Drive
Kanata, Ontario, Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 2.1 (1), 2.2 (1), 2.6 (1..6), 2.10 (4), 2.10 (7), 3.1 (7), 3.5.2 (1..4), 3.6.3 (3), 4.2 (3..5), 4.5.2 (9), 4.5.3 (2), 14, Appendix C

PROBLEM:

In Canada operational software used by the Department of National Defence (DND) must provide an equivalent user interface in each official Canadian language: English and French. Consequently, the displayed text and accepted inputs for an interactive program must be equally clear and usable in both English and French. The user must be able to select the language that will be used at the beginning of each interactive session.

These programs are difficult to develop because the Ada standard provides direct support for English, but does not provide equivalent support for French. The predefined type CHARACTER does not include the accented characters required for French. String literals cannot easily include accented French characters, and the string comparison operators such as "<" do not sort French words correctly.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

1. Use the predefined CHARACTER and STRING types and output French words without accents. This creates difficulties for the French user. For example, the French work "cote" can have four different meanings depending on the accents used. This approach is not acceptable for critical DND applications.

2. Define a new type and provide the required operations for input, output, string comparison, and other functions. The type and its operators can be placed in an Ada package. However, with this approach it is difficult to express string literals for the new type.

POSSIBLE SOLUTIONS:

Base the Ada standard on ISO 8859-1 instead of ASCII.

HANDLING OF LARGE CHARACTER SET IN ADA

DATE: October 2, 1989

NAME: Yoneda Nobuo

ADDRESS: Department of Information Science
Tokyo University
7-3-1 Hongo Bunkyo-ku
Tokyo 113 Japan

TELEPHONE: +81-3-812-2111
+81-3-818-1073 (fax)
E-mail: nishida@nttslb.ntt.jp (CSNET)

ANSI/MIL-STD-1815A REFERENCE: 2.1, 2.5, 2.6, 2.7, 3.5.2, 3.5.5, 3.6.3, 4.2, 4.6, 14.3, 14.4, Appendix A, C

PROBLEM:

Current Specification does not allow using of multi-octet character set.

With relatively recent advances in desktop publishing and international communication, it is increasingly important for programming languages to provide guarantees for text processing applications in an international level. In this regard, ISO SC22 asks it working groups to take necessary actions to internalize their languages especially in character handling features.

Ada limits the characters in the program text to the ASCII graphic characters and this makes it prohibitive, rather than simply inconvenient, for us to handle Japanese text, which uses large set of characters. It is highly desirable to modify Ada so that a large character set can be handled easily in it.

In accordance with DoD's Ada 9x project, we propose modification of Ada in character handling to make it truly internationalized language.

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

See Handling of Large Character Set in Ada, SC22/Ada WG Japanese Body, September 12, 1989.

PRONUNCIATION OF SYMBOLS

DATE: October 30, 1989

NAME: Jolie Mason

ADDRESS: Unisys Corporation
Mail Station B205
5151 Camino Ruiz
Camarillo, CA 93010

TELEPHONE: (805) 987-6811 Ext. 4582

ANSI/MIL-STD-1815A REFERENCE: 2.1, 2.2

PROBLEM:

Chapter 2 gives official names for special characters, other special characters and compound delimiters. In a number of cases, these do not correspond to common usage or are prone to error in (human or computer- synthesized) voice communication.

The Ada language was designed with a concern, explicitly stated in the Steelman requirements, that reserved words should be pronounceable. Common practice in the Ada community is to also make identifiers pronounceable. It is logical to extend this concern to the Ada symbol set as well.

Making the Ada symbols as readable as possible would not require a change to the Ada language, but it would require a change to the Reference Manual.

There are several reasons why it is important to make such a change.

(1) There is a growing number of blind Ada programmers, and to this group a readable language means a hearable language. A blind programmer must be able to completely understand Ada code by listening to the code as it is read.'

(2) There is new federal legislation (Public Law 99-506 Section 508, "The Electronic Curb-Cut Law") which requires all computer equipment purchased or leased by the federal government to be adaptable for the handicapped. This means that all government-related computer hardware, software and databases must be adaptable for the handicapped.

(3) The most important reason is that everyone, at one time or another, must communicate Ada verbally. We all need to communicate verbally as we discuss problems, solutions or questions regarding code. The increasing use of the telephone and teleconference makes it more and more important that we speak Ada consistently. Spoken communication is not as efficient if it requires negotiating how symbols are to be referred to in order to insure unambiguous communication.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: N/A

POSSIBLE SOLUTIONS:

The following suggested pronunciations are the result of evaluating the pronunciation choices used by three distinct voice synthesis speech sets, the reading instruction manual used by Recording for the Blind, and Verbal Mathematics by Author.

In summary, the research showed that voice selections varied between synthesizers. Common usage is arbitrary; however, the choices in the suggested pronunciations prefer a strong internal consistency. It is also important to note that the more quickly the pronunciation of the symbol can be finished, the easier it is to understand; for example, the longer word "parenthesis" is shortened to "paren." The Recording for the Blind manual makes a convincing case for distinguishing punctuation from text; for example, the word "sign" has been added after some symbol names. The mathematical pronunciation guide supports scientific choices for pronunciation, rather than linguistic or musical references; for example "caret" is used rather than "circumflex."

symbol	name	suggested name
"	quotation	double quote

The symbol name remains consistent whether or not preceded by "double".

#	sharp	number sign, pound sign

The musical reference is inappropriate in the Ada context.

&	ampersand	ampersand, and sign

The meaning of the word "and" is self-evident; the word "sign" is must be included to distinguish text from punctuation.

'	apostrophe	single quote, prime

The word "single" must be included because the word "quote" is frequently used to mean either " or '.

(	left parenthesis	open paren

The phrase "paren" is completely understandable; the longer phrase "parenthesis" takes unnecessary time.

"Open" and "close" make pairing clear. For example, "left paren left paren expression right paren left paren expression right paren right paren" is confusing; "open paren open paren expression close paren open paren expression close paren close paren" is easier to process mentally.

)	right parenthesis	close paren

See: (

*	star, multiply	

+ plus

, comma

- hyphen, minus dash, minus

The symbol name remains consistent whether or not preceded by "double", since "double dash" is used for comments.

. dot, point, period

: colon

; semicolon

< less than less than, open angle bracket

"Open" and "close" make pairing clear when used as angle brackets.

All bracket symbol characters have consistent names of the form <action> <type of line used to draw symbol> bracket.

= equal

> greater than greater than, close angle bracket

See: <

_ underline underscore, under

The symbol is used as a character of itself and is not used to underline another character.

"Under" is shorter but perfectly clear.

| vertical bar vertical line, or sign

The word "line" is used in other symbols.

The word "or" is meaningful when used in choices; "sign" must be included to distinguish text from punctuation.

! exclamation mark exclamation mark, exclamation

"Exclamation" is shorter but perfectly clear.

$ dollar dollar, dollar sign

"Dollar sign" is common usage.

% percent

? question mark question mark, question

"Question" is shorter but perfectly clear.

@ commercial at at sign

The phrase "commercial" is not commonly used.

[left square bracket open square bracket

"Open" and "close" make pairing clear.

All bracket symbol characters have consistent names of the form <action> <type of line used to draw symbol> bracket.

The word "square" must be included because the word "bracket" is frequently used to mean either [,] or {, }.

\ back slash

] right square bracket close square bracket

See:]

^ circumflex caret, up arrow

The linguistic reference is inappropriate in the Ada context.

` grave accent grave accent, back accent

The linguistic reference is inappropriate in the Ada context.

{ left brace open curly bracket

"Open" and "close" make pairing clear.

All bracket symbol characters have consistent names of the form <action> <type of line used to draw symbol> bracket.

To eliminate the confusion between "bracket" and "brace," only the word "bracket" is used to describe bracket or brace symbols.

} right brace close curly bracket

See: {

~ tilde

=> arrow association, arrow,

associated with

The meaning of the sign is more clearly understood if it is included in the pronunciation of the symbol.

.. double dot

** double star,
exponentiate

:= assignment
(pronounced: "becomes")

/= inequality
(pronounced: "not equal")

>= greater than or equal

<= less than or equal

<< left label bracket open label bracket

"Open" and "close" make pairing clear.

>> right label bracket close label bracket

See: <<

<> box

USE OF PARENTHESES FOR MULTIPLE PURPOSES

DATE: October 30, 1989

NAME: Stephen J. Schmid

ADDRESS: Hughes Aircraft Company
Building S31-P322
P.O. Box 92919
Los Angeles, CA 90009

TELEPHONE: (213) 648-2098

ANSI/MIL-STD-1815A REFERENCE: 2.1, 3.6, 4.6, 6.1

PROBLEM:

In Ada, parentheses are used for all of the following:
-Enclosing parameters to a subprogram
-Indication an array indexing operation
-Enclosing an expression for a type conversion
When an Ada statement is written which contains more than one of these operations, (and especially if it contains all three), it becomes very unreadable, due to the lack of clarity of the function of the parentheses in each case.

Example:

```
A := function_name (array_name (type_name (parameter)));

A := B ( C ( D ( E ) ) );
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Choosing names wisely helps somewhat, but does not solve the problem.

POSSIBLE SOLUTIONS:

I feel the best solution would be the following:
Use square brackets for array indexing.
Use braces for type conversion.
Compilers could be written to either accept both forms (for compatibility with past programs), or a switch could be provided to do one or the other, or, a conversion program could be provided which would change existing code.

Then the above example would be:

```
A := B ( C [ D ( E ) ] );
```

Even without descriptive names, we can now tell what is going on in this statement.

MAKE EVERY BIT AVAILABLE TO THE APPLICATION PROGRAMMER

DATE: October 27, 1989

NAME: Michael F. Brenner

DISCLAIMER:

The opinions are mine; not necessarily InterACT's.

ADDRESS: InterACT Corporation
417 Fifth Avenue
New York, New York 10016

TELEPHONE: (212) 696-3680

ANSI/MIL-STD-1815A REFERENCE: 2.1(1), 8.6, 4.5, 3.5.4, 13

PROBLEM:

Address clauses placing data at address 16#FF_FF_FF_FF# on a 32 bit machine, bit strings extending beyond a word boundary in a record representation, 32-bit unsigned integers for 32 bit processors, full range of fixed point numbers, the last bit of accuracy in floating point numbers, and eight-bit ASCII have been submitted in various forms to the Ada Commentaries and to the Ada 9X Project as if they were unrelated problems.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

The current workaround is for these to be considered by the Ada 9X Committee as if they were separate unrelated issues, with the consequence that a non-integral approach might be accepted.

POSSIBLE SOLUTIONS:

The Ada 9X Committee should consider these problems as a single integrated issue and remove any restrictions the language imposes which prevent an embedded systems developer from using every available bit.

INCOMPATIBLE NATIONAL VARIATIONS OF THE ISO STANDARD 646

DATE: October 16, 1989

NAME: C. Gregor H. Stenderup

ADDRESS: Aarhus University
Computer Science Department
Ny Munkegade 116
DK-8000 Aarhus C
Denmark

TELEPHONE: +45 86 12 71 88
E-mail: gregor@daimi.dk

ANSI/MIL-STD-1815A REFERENCE: 2.1(1), 2.1(4), 2.1(10), 2.1(13), 14.3.10(1), C(13), C(15)

PROBLEM:

The incompatible national variations of the ISO standard 646 make it impossible for Ada to support the modern European text-handling requirements.

These national variations are not merely that some special characters have been given different graphical representations (as 2.1(13) seems to indicate when comparing the US and the UK variations), but also that many countries have had to discard some special characters in order to make room for all their letters. The national variations of ISO standard 646 have therefore become incompatible. As an example let me show the letters of Denmark and Germany.

Denmark:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
	a b c d e f g h i j k l m n o p q r s t u v w x y z
Germany:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
	a b c d e f g h i j k l m n o p q r s t u v w x y z

Both Denmark and Germany have discarded '[', '\', ']', '[', '{' and '}' in order to give the letters consecutive ordinal order.

Currently the EEC is doing a lot of work to merge the independent home markets of its twelve memberstates into one huge international market. This work is to be completed by 1992.

As a consequence there will be applications that need to store or handle information originating from more than one country. Such applications include maintaining a European mailing-list (names, addresses, ...), automatic translation of text (laws, directives, regulations, letters,...) between any two of the languages used in the EEC, electronic document interchange within the EEC,...

None of these applications can accept having to choose one of the national variations of the seven-bit ISO standard 646, they all need an eight-bit version of the ASCII standard.

The AJPO might also face this problem, if an Ada-program is used to record this Ada 9X revision request.

IMPORTANCE: ESSENTIAL

You cannot expect the european countries to discard parts of their alphabets in order to accommodate application design in Ada. If the request is not satisfied by the revision, it is more likely that other languages (Pascal, Modula-2,...) would be used for most application design in Europe (the UK could be an exception).

CURRENT WORKAROUNDS:

A workaround could be to redefine the type CHARACTER to have 256 elements or to map the characters onto bytes as done in C. Input and output of characters from/to a terminal/console can then be done with the SEQUENTIAL_IO package. Both SEQUENTIAL_IO and DIRECT_IO can also be used with disk files. However TEXT_IO cannot be used with such a redefined CHARACTER type.

POSSIBLE SOLUTIONS:

Replace the seven-bit ISO standard 646 with a suitable eight-bit version of the ASCII standard, thus changing C(13):

```
FOR CHARACTER USE (0, 1, 2, ..., 254, 255);
```

Logically the CHARACTER type belongs in the ASCII-package (C(15)), into which it should be inserted. STANDARD should then rename it:

```
TYPE CHARACTER RENAMES ASCII.CHARACTER;
```

Eventually a pragma ASCII could be used to replace the ASCII-package (some hosts support several ASCII-versions):

```
PRAGMA ASCII (version);
```

Another possibility would be to make the TEXT_IO-package another package (or an instance of a generic package). This would also solve the racing problems that can occur when using TEXT_IO and file variables inside TEXT_IO's body are shared variables).

PROPOSAL FOR AN EXCHANGE OPERATOR

DATE: October 22, 1989

NAME: Douglas Arndt

ADDRESS: SAIC
5151 East Broadway
Suite 900
Tucson, AZ 85711

TELEPHONE: (602) 748-7400

ANSI/MIL-STD-1815A REFERENCE: 2.2, 5

PROBLEM:

Exchanging values between two variables is commonplace but using traditional Ada techniques is clumsy and potentially inefficient on newer architectures.

IMPORTANCE: IMPORTANT

This would be a very nice feature, particularly in applications where exchanges are common such as matrix manipulation.

CURRENT WORKAROUNDS:

Typically, a temporary variable is used in the following fashion:

```
temp := a;
a    := b;
b    := temp;
```

This is not particularly attractive or readable since it takes three assignments to accomplish one logical operation. The operation can be encapsulated in a generic procedure but this requires an instantiation everywhere it is used. Furthermore, this process is inefficient on newer architectures that support exchanges directly in hardware or microcode.

POSSIBLE SOLUTIONS:

A new operation for exchanges should be added to the language. Add a new compound delimiter, ":=:" (called "exchange"), to LRM 2.2. Add a section to chapter 5 describing the exchange statement. The exchange should be pre-defined for all objects that are assignment compatible. The code example given above could then be rewritten:

```
a :=: b;
```

Note: the exchange operator in the form given above (i.e., ":=:") was adapted from the Icon language.

CLUMSY SYNTAX FOR REPRESENTING BASED NUMBERS

DATE: September 23, 1989

NAME: Stephen J. Wersan, Ph.D.

ADDRESS: Code 3561
NAVWPNCEN
China Lake, CA 93555

TELEPHONE: (619) 939-3120,
Autovon 437-3120
E-mail: WERSAN%356VAX.DECNET@NWC.NAVY.MIL

ANSI/MIL-STD-1815A REFERENCE: 2.4.2, 4.3

PROBLEM:

Clumsy syntax for representing based numbers, especially when used in an aggregate.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Stick to established syntax.

POSSIBLE SOLUTIONS:

Suggested revision to aggregate syntax --

An entire aggregate or a subunit of an aggregate which must currently be written as shown in the following examples

```
(8#275#, 8#276#, 8#277#) or
(RED => 8#275#, PINK=> 8#276#, BLUE => 8#277#)
```

could also be written as follows

```
8#(275, 276, 277)# or
8#(RED=> 275, PINK => 276, BLUE => 277)#
```

One might regard the construction '8#(' as opening a declaration (of assumed radix) whose scope is ended by ')#', so that nesting one such declaration inside another may be treated by the ordinary rules of scoping and visibility.

MEANING OF PRAGMA'S NOT IMMEDIATELY OBVIOUS

DATE: October 12, 1989

NAME: William Thomas Wolfe

ADDRESS: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: (803) 656-2847
E-mail : wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 2.8

PROBLEM:

The keyword "pragma" is not a part of the English language, and its meaning is not immediately obvious.

CONSEQUENCES:

New users of Ada find this keyword to be non-intuitive, contributing to the difficulty of learning and using Ada.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Substitute "Compiler:" for "pragma".

To show that this is more natural, apply the following test:

Below is a list of keywords from one or more unknown programming languages. Please write down your best guess as to the purpose of each keyword.

1) pragma

2) Compiler:

If this test is given to a programmer who is reasonably intelligent but has no prior knowledge of Ada, the results will be as follows:

1) Don't know.

2) This looks like a compiler directive.

This sort of procedure should generally be followed when selecting keywords, in order to minimize the difficulty of learning and using the language and thereby maximize the language's acceptance.

LEGALITY OF PROGRAMS WITH IMPL.-DEFINED PRAGMAS

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 2.8

PROBLEM:

There are numerous examples of implementation-defined pragmas whose purpose is a user-provided guarantee to adhere to certain restrictions. E.g., tasking related pragmas that promise certain characteristics of an entry.

It is counter-productive to require that the compilation must succeed even if such assertions are violated. The user should be warned as early as possible (by non-acceptance of his/her program) that the given assertions have been violated.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

2.8 (8) should be altered to allow for implementation-defined pragmas that may render text outside such pragmas illegal. However, no Ada implementation may require the presence of such pragmas in Ada programs.

A FACILITY TO TURN OFF OPTIMIZATION

DATE: October 12, 1989

NAME: B. A. Wichmann (endorsed by Ada UK)

ADDRESS: National Physical Laboratory
Teddington, Middlesex
TW11 OLW. UK

TELEPHONE: +44 1 943 6076 (direct)
+44 1 977 3222 (messages)
+44 1 977 7091 (fax)
E-mail: baw@seg.npl.co.uk

ANSI/MIL-STD-1815A REFERENCE: 2.8 and Appendix B

PROBLEM:

The language defines a pragma to optimize, either for time or space. However, no facility is provided in the language to turn off optimization, should an implementation perform optimization by default.

If optimization were always correct and did not increase compiling speeds significantly, then there would be little need to have the requested option. However, research at NPL has indicated that even mature compilers for simpler languages than Ada contain significant bugs in the code-generator, and that turning off any optimization reduces the bugs. The UK Interim Defense Standard on safety-critical software (00-55) specifies that optimization should be turned off.

In most critical applications in which bugs are unacceptable, the user will wish to turn off optimization even if some performance loss is incurred. This can only be achieved by means of a new language-defined pragma in a portable fashion.

Another situation in which removal of optimization may be needed is concerned with debugging, since otherwise there is likely to be no simple correspondence between the source text and the binary program. Similarly, this simple correspondence may be needed if for assurance reasons, the output from the compiler must be checked by hand.

IMPORTANCE: IMPORTANT

for some critical application areas.

CURRENT WORKAROUNDS:

An implementation can either not perform optimization by default, or provide an implementation-defined pragma to turn off optimization. An alternative approach is for an implementation to provide a different mechanism, outside the language, to control the optimization. An example of this approach is a compiling option invoked as a parameter to the command line on calling the compiler.

POSSIBLE SOLUTIONS:

Add a new language-defined pragma to the language or extend the existing pragma OPTIMIZE.

REQUIRED WARNINGS FOR UNRECOGNIZED PRAGMAS

DATE: October 31, 1989

NAME: Mike Kamrad

ADDRESS: Unisys Computer Systems Division
M/S U2F13
PO Box 64525
St. Paul MN 55164-0525

ANSI/MIL-STD-1815A REFERENCE: 2.8

PROBLEM:

The Ada language does not require an implementation to warn the user about unrecognizable pragmas. Failure to warn the user about an unrecognizable pragmas can mislead the user. All pragmas which the implementation can not recognize a pragma should produce a warning to the user.

IMPORTANCE:

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

All pragmas which the implementation can not recognize a pragma should produce a warning to the user.

CURRENT WORKAROUNDS:

JUSTIFICATION/EXAMPLES/WORKAROUNDS:

The user may never learn that a pragma is not recognized by an implementation until execution time. This will cause more debugging work for Ada users for a situation that an implementation can easily recognize at compile time. Supposedly the Ada user can consult Appendix F of the Ada language reference manual to find out which pragmas is not recognized. Of course, if this same reasoning could be used for all other syntactic and semantic rules of Ada, then there would be no need for any warning or error messages from any implementation.

NON-SUPPORT IMPACT:

This will cause more debugging work for Ada users for situation that an implementation can easily recognize.

POSSIBLE SOLUTIONS:

All pragmas which the implementation can not recognize a pragma should produce a warning to the user.

DIFFICULTIES TO BE CONSIDERED: NONE

REFERENCES/SUPPORTING MATERIAL: NONE

REPORTING OF PRAGMA ERRORS

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 2.8(9, 11)

PROBLEM:

A compiler is not required to report an unrecognized or incorrectly used (including parameter usage and placement) pragma.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Since the compiler is checking these conditions anyway, the compiler should report the problems found, i.e., unrecognized, misplaced, incorrect, etc. pragmas and use.

REQUIRE WARNINGS FOR PRAGMAS IGNORED

DATE: October 20, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 2.8(11)

PROBLEM:

In paragraph 2.8(11), the LRM suggests that compilers issue warnings when pragmas are ignored. This should be made a requirement for two reasons. First, this allows a badly placed pragma to be ignored with no indication given to the programmer; the rules for pragma placement are not well-formed (different pragmas have different rules; the syntactic category pragma does not appear elsewhere in the syntax rules of the language); this puts a burden on the programmer to verify pragma placement. Second, some pragmas (such as INLINE and PACK) may have important effects on the performance/size of the resulting code: it seems desirable to know whether or not the compiler intends to obey them.

IMPORTANCE: *IMPORTANT*

CURRENT WORKAROUNDS: Not applicable

POSSIBLE SOLUTIONS:

Require that compilers indicate whenever a pragma is ignored and the reason (e.g., badly placed, syntax or other error in the pragma, or not recognized by the current implementation).

DO NOT ADD NEW RESERVED WORDS TO THE LANGUAGE

DATE: August 31, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 2.9

PROBLEM:

Existing programs may have identifiers for, e.g., types, that duplicate new keywords. Migrating such programs to Ada 9X will be costly.

IMPORTANCE: ESSENTIAL

Otherwise the cost of migrating to Ada 9X may be unacceptable.

CURRENT WORKAROUNDS:

Rewrite all existing compilation units that have identifiers matching reserved words in Ada 9X.

POSSIBLE SOLUTIONS:

Ensure that all new syntactic categories are such that keywords can be distinguished from identifiers by context.

ELIMINATION OF REPLACEMENT CHARACTERS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706[11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 2.10(1-8)

PROBLEM:

The three replacement characters '!' for ' ', ':' for '#' and '%' for '"' make listing much harder to read, as well as increasing the complexity of parsers.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Don't use the replacement characters.

POSSIBLE SOLUTIONS:

Eliminate the three replacement characters. Use the normal ASCII character set instead.

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed. However, since almost every everyone has printers that print the full ASCII character set (this being 1989 instead of 1946, after all), almost nobody uses the replacement characters, thus, this change will impact very few people, if any.

For additional references to Section 2. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 3. DECLARATION AND TYPES

IMPORTANCE: ESSENTIAL

This is no way of overcoming this problem, without using assumptions about the type about to be made visible, and probably unchecked conversions.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Allow for the limitations of order of declaration introduced by generic instantiations, and permit wider use of incomplete types, possible in a similar way to private types prior to their full declaration.

INITIALIZATION FOR ALL DATA TYPES

DATE: March 21, 1989

NAME: Larry Langdon

ADDRESS: Census Bureau
Room 1377-3
Federal Office Bldg 3
Washington, DC 20233

TELEPHONE: (301) 763-4650
E-mail(temporary): langdonl@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3

PROBLEM:

It seems peculiar and makes the language feel asymmetric when you can create a record type that carries its own initialization with it but you can't make (for example) an integer type that does likewise. Initialization should be allowed for all non-limited data types. Examples of the use of this feature include:

a) The creation of an integer type whose objects' values are always initialized to zero unless another value is specified. In particular, these objects are always initialized.

b) The creation of a string type whose objects are always initialized to blanks in the aosence of another specific initialization.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Depending on the situation, using initializations in object declarations, or changing a type declaration to have an "enclosing" record.

POSSIBLE SOLUTIONS:

Acceptable syntax for this feature might include:

```
type COLOR is (white, red, yellow, green) := yellow;
type my_integer is range 1..100 := 33;
type int0 is new integer := 0;
type strblank is new string := (others=>' ');
```

NOTE: This proposal (in slightly different, but substantively identical form) was approved as an Ada Language Issue (LI58) by the Ada Language Issues Working Group of SIGAda. The final vote, taken March 1, 1989, was:

9 in favor
3 against

PROPOSAL FOR SUBPROGRAM TYPES

DATE: October 21, 1989

NAME: Douglas Arndt

ADDRESS: SAIC
5151 East Broadway
Suite 900
Tucson, AZ 85711

TELEPHONE: (602) 748-7400

ANSI/MIL-STD-1815A REFERENCE: 3, 6

PROBLEM:

There is currently no way to specify subprograms as objects. In particular, they can't be assigned to variables, stored in array and record structures, or passed as parameters to other (non-generic) subprograms.

The lack of syntax and semantics for subprogram types creates serious problems implementing a certain class of software component that I will call "dispatchers". Dispatchers are a necessary part of many applications. I have tried to capture the essence of the problem in the following code:

```
package Menu_Example is

        --      This package is a simple abstraction of a general-purpose menu dispatcher.  It is very
        --      similar to menu interfaces provided on some Unix workstations.
------

        type MENUS is private;
        --      MENUS objects can consist of any number of "buttons".  A button has a name and an
        --      associated action.  Buttons can be dynamically added and removed from a menu item.
------

        type BUTTON_IDENTITY is ...;  -- whatever; probably dynamic string

        procedure Add_Button  (to_the_menu : in out MENUS;
                                                button_name: in     BUTTON_IDENTITY;
                                action_to_perform : in     ?);

        procedure Push_Button (the_menu    : in MENUS;
                                button_name : in BUTTON_IDENTITY);

private

        type MENUS is ...

end Menu_Example;
```

The specification is properly abstracted from a software engineering viewpoint. That is, the implementation of the MENUS type is hidden and the package doesn't have unnecessary visibility into any higher-level units. The problem now is how to specify the action to be taken when a button is pushed. The most straight-forward way is to pass the procedure to the constructor routine (i.e., Add_Button) for storage with the menu object until the button is pushed.

```
package Menu_Example is

    type MENUS is private;

    type BUTTON_IDENTITY is ...;

    type ACTION is procedure;
    --              ***^^^^^^^^^^***
    --      Note that this is the declaration of a subprogram type. The exact syntax is TBR.
  -----

    procedure Add_Button (to_the_menu    : in out MENUS;
                          button_name    : in     BUTTON_IDENTITY;
                       action_to_perform : in     ACTION);
    --      ***^^^^^^^^^^^^^^^^^^^           ^^^^^^^***
    --      action_to_perform is now a subprogram of type ACTION that can be invoked or assigned
            to a variable within Add_Button
    -----

    procedure Push_Button (the_menu      : in MENUS;
                           button_name   : in BUTTON_IDENTITY);

private

    type MENUS is ...

end Menu_Example;
```

Now a user can add buttons to a menu dynamically and designate the procedures associated with each. Assuming that the implementation of MENUS objects is an array of buttons, and each button is a composite type with fields for the name and action, then the body of Add_Button might look like:

```
    procedure Add_Button (to_the_menu    : in out MENUS;
                          button_name    : in     BUTTON_IDENTITY;
                    action_to_perform    : in     ACTION) is
        begin
                ...      -- determine the index of the next button
                to_the_menu (i).name_field := button_name;
                to_the_menu (i).associated_action := action_to_perform;
                -- ^note: assigning a subprogram to a field in a record object
                ...
        end Add_Button;
```

The body of Push_Button would be able to access the same field, and, since it is a subprogram, execute it directly as it would any other:

```
procedure Push_Button (the_menu    : in MENUS;
                       button_name : in BUTTON_IDENTITY) is
begin
        ... -- search through all the buttons via loop, table lookup
        -- or whatever
        if the_menu (i).name_field = button_name then
        the_menu (i).associated_action;  -- procedure call!
        end if;
end Push_Button;
```

IMPORTANCE: ESSENTIAL

A wide variety of languages, including C, Modula-2, Lisp, many Pascals, and even Fortran, provide some form of subprogram typing, allowing them to be passed as actual parameters and/or stored as variables. The lack of a comparable feature in Ada may (and, I believe, will) cause many to ignore Ada's positive features and select another language.

BiiN felt that the need for subprogram types was so important that they went "outside" the language and implemented them via pragmas. Unless subprogram types are added to Ada 9x, I fear that there will be more attempts to extend the language in such uncontrolled ways.

CURRENT WORKAROUNDS:

The universal nature of dispatchers makes them necessary in many applications. The Ada research literature is beginning to see more references to the difficulty of implementing them in Ada. One description is found in "ERS: An Expert System Shell Designed and Implemented In Ada" by Stuart Hirshfield and Thomas Slack (Proceedings of AIDA-88, published by George Mason University). They analyze the problem (specifically, the inability to implement jump tables) in detail and discuss different workarounds and why none are completely acceptable. They state:

> "...Ada's lack of a funcall equivalent caused us some inconvenience. In terms of ERS, the easiest way to dispatch to the evidence and external action functions is to place the actual function pointers into the inference net where they will be needed (i.e., within the nodes they are associated with). A funcall equivalent could then be used to execute them directly."

Hirshfield and Slack discovered a VAX-specific way to pass and execute subprogram objects by calling operating system interface routines. They showed uncommon good sense and restraint by rejecting that approach and working harder to find an Ada solution. The eventual solution was not very good from a software engineering perspective, however, and would undoubtedly be scoffed by C programmers: they wrote a program to generate a package with hard-coded options.

Generics do not offer an acceptable solution for a number of reasons. First of all, they lack a mechanism to store the subprograms passed to them as parameters. That is, a subprogram passed to a generic unit is only visible within that unit and can't be made available to other units via assignment to global structures. Furthermore, it is not possible to pass a variable number of arguments to generic units. Finally, generic instantiations exist from the time of their instantiation until the end of their declarative scope. It is not possible to selectively "de-instantiate" them without the use of complex scoping techniques.

POSSIBLE SOLUTIONS:

The example given above demonstrates a possible solution where the keyword "procedure" is used within a type statement. Alternately, the syntax could be adapted from task type declarations, e.g.,

```
function type IS_IT_PRIME (number : INTEGER) return BOOLEAN;
              ^ ^ ^ ^ ^
```

The specific syntax is really unimportant as long as subprograms (both procedures and functions) can be assigned to variables, aggregated within arrays and records and passed as parameters to other subprograms.

Any approach that is adopted should be consistent with Ada's historic goals, especially strong typing. I see no reason why the addition of subprogram types can't be completely upward compatible with existing Ada.

NEED "SEQUENCE" TYPE WITH FIXED LOWER BOUND

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3

PROBLEM:

The fact that Ada supports arbitrary lower bounds for array types which are representing sequences rather than mappings is a serious source of bugs in Ada code.

It is very common to simply code "str(1)" instead of "str(str'FIRST)" and to code "str(str'LENGTH)" instead of "str(str'LAST)". This kind of code introduces bugs which only reveal themselves long after the code is thought to work perfectly, when by chance the "str" parameter happens to have a lower bound other than 1 (because it is a slice, usually).

Ada provides automatic array subtype conversion (aka "sliding") to partially overcome the problems associated with explicit lower bounds, but the contexts where sliding is performed are not well know to most Ada programmers, and the presence of sliding makes named array aggregates with an others clause illegal in the same contexts to the great mystery of most Ada programmers.

When an array is in fact representing a mapping from the index type to the operand type, sliding is generally an error (especially if the index type is an enumeration type), however there is no way to declare that sliding is illegal, especially on assignment, meaning that non-meaningful assignments can be performed.

The fundamental problem is that Ada uses the single array class of types to represent both sequences, where explicit lower bounds are a pain and "sliding" should be automatic, and mappings, where an explicit lower bound is often useful and sliding is not generally meaningful.

IMPORTANCE: ESSENTIAL

Confusions over sliding, and 'LAST /= 'LENGTH, 'FIRST /= 1, are a major source of bugs, introduced by Ada's explicit arbitrary lower bounds for arrays.

CURRENT WORKAROUNDS:

The workaround for the named-with-others array aggregate problem is generally to enclose it in a qualified expression. The workaround to prevent sliding on assignment is to turn the array type into a discriminated record type with a nested array. Unfortunately, this eliminates the built-in concatenate and slicing.

POSSIBLE SOLUTIONS:

It should be possible to distinguish between "mapping"-style arrays and "sequence"-style arrays. For mapping arrays, no automatic sliding should occur, though explicit subtype conversion should probably be allowed.

For sequence arrays, the lower bound should be fixed at 1, with all operations (concatenate, slice, string literal) producing a result with a lower bound of 1. Index constraints for sequences should specify only the length/'LAST, and not 'FIRST.

The simplest solution is to provide two pragmas

pragma MAPPING(**<array type>**);

and

pragma SEQUENCE(**<array type>**);

The effect of pragma MAPPING would be to disable automatic sliding, and to allow the use of named aggregate with others in all contexts where the bounds of the result are known from context.

The effect of pragma SEQUENCE would be to allow the use of a single high bound in an index constraint, to require that the low bound be 1 if specified, and to cause all slices and concatenate to automatically slide so that the resulting sequence has a low bound of 1 again. 'FIRST would be statically equal to 1 for all objects of a sequence type, including formal parameters of "unconstrained" sequence type. The low bound of the index subtype in an array sequence type definition would be required to be statically equal to 1. Note that this would mean that the "dope" for an unconstrained sequence parameter would only need to include the high bound, thereby speeding up parameter passing, and all component references within the subprogram.

Ada 83 rules would apply in the absence of either pragma, and for all multi-dimensional array types. STANDARD.String would be declared to be a SEQUENCE.

ALLOW DISCRIMINANT OF ARBITRARY NON-LIMITED TYPE

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3

PROBLEM:

Ada currently restricts record-type discriminants to discrete types. There is no apparent reason for this restriction. Since discriminants are the ways that a type may be parameterized, it is useful to be able to provide any non-limited type of parameter.

Furthermore, discriminants may only be changed by full-object assignment, and therefore represent effectively "constant" components. There are times when it is important that a component of a record be treated as a constant, but currently, this is only possible for discrete types.

IMPORTANCE: IMPORTANT

When declaring an object, it is often useful to parameterize the declaration/initialization, with any type of discriminant. Furthermore, it is desirable to be able to have constant components of a record of any type.

CURRENT WORKAROUNDS:

To parameterize an initialization, it is possible to pass any type of object to a function used to initialize the object. However, there is no way to control default initialization this way.

POSSIBLE SOLUTIONS:

Allow any non-limited type as a discriminant.

ALLOW INITIALIZATION/FINALIZATION FOR TYPES

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3

PROBLEM:

Only certain classes of types allow explicit initialization. Furthermore, the initialization is limited to expressions initializing components of a record. No types allow explicit finalization. This is useful for types which require some resource deallocation or release prior to leaving the scope in which an object is declared. This would make the most sense for limited types, since other types use built-in assignment for copying.

IMPORTANCE: ESSENTIAL

Generalized initialization, and finalization for limited types is critical to being able to construct safe and efficient subsystems.

CURRENT WORKAROUNDS:

Initialization can be accomplished by wrapping a type in a record.

There is no support for finalization, and it requires that all users of a type make an explicit call to a cleanup subprogram before exiting a scope.

POSSIBLE SOLUTIONS:

All types should allow the specification of a default initial expression. Private types should allow the specification of a sequence of statements to initialize the object.

Limited types should allow the specification of a procedure to be called upon scope exit, or unchecked deallocation of an object of that type. Limited types with finalization are treated somewhat like task types when returned from a function, namely the returned object is "not" a copy, but rather designates the same object. It is erroneous to return an object outside of its scope. Discuss possible solutions for addressing the stated problem.

SUPPORT EXPLICIT REFERENCES TO OBJECTS

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3, 6

PROBLEM:

There are times when it is useful to create an explicit reference (pointer) to an object. Currently, an object rename in Ada effectively creates a reference to an object. However, it is not possible to return a reference to an object. This means that it is not possible to implement an array abstraction in Ada where a function could return a reference to a particular element of an array available for assignment.

Furthermore, there are times when it is semantically relevant whether an object is passed by copy or by reference. It is generally easy to force pass by copy, by simply declaring a local copy. However, there is no way to force pass by reference. This is particularly important for shared objects.

IMPORTANCE: IMPORTANT

A good general reference mechanism can be used to solve many problems, including passing subprograms as parameters, controlling copying of shared objects, and implementing an array abstraction.

CURRENT WORKAROUNDS:

Renames provide a limited form of reference, but they cannot be used for returning an object, nor do they ensure parameter passing by reference.

POSSIBLE SOLUTIONS:

There are really two problems, one is passing and returning references to objects to/from subprograms. The other is storing references to objects (or subprograms) for later use.

When passing a reference to an object, there is generally no danger that the object will disappear before the reference is gone, so lifetime of the reference is not a big issue.

When returning a reference, generally the reference is either to a global, or is to some component of one of the parameters of the subprogram.

For stored references, the reference-type should be declared much like an access type, except that instead

of an allocator, references are created by converting from a designated object. The conversion is only allowed if the scope of the designated object ends at the same point, or later than that of the reference type. This means that the designated object must be declared at the same level as the reference type, or must be declared in a declarative region enclosing the declaration of the reference type. Typically, both the reference type and the designated objects

ADA SHOULD SUPPORT INHERITANCE AND POLYMORPHISM

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3

PROBLEM:

Inheritance and polymorphism as supported by languages such as C++ and SmallTalk represent an important mechanism for disciplined evolution and reuse of code.

Ada currently has minimal support for inheritance (via derived types), and no support for run-time polymorphism (the ability for code to take objects of more than one type and to automatically select appropriate operations based on the run-time type). This lack of support interferes with the use of Ada in areas of growing importance such as object-oriented graphic interfaces, object-oriented databases, type-safe extensible interpreters, etc.

IMPORTANCE: ESSENTIAL

If Ada is not augmented to support inheritance and polymorphism, an important class of problems will be significantly more difficult to solve in Ada than in languages like C++. Furthermore, code reuse will be limited since existing abstract data types cannot be reused for slightly different applications without copying and editing the source, which is often impractical due to proprietary code or configuration management considerations.

CURRENT WORKAROUNDS:

Heavy use of generics can sometimes work around the limitations. However, the resulting generics present major stress to current compiler technology, and rarely achieve the flexibility or reusability desired.

POSSIBLE SOLUTIONS:

As a minimum for inheritance, it should be possible to define a derived record type which has more components than the parent type, as follows:

```
type T1 is new T0 with record
   F1 : integer := E1;
   F2 : float := E2;
end record;
```

This could be preceded by "type T1 is new T0 with private;" in a visible part if desired. The resulting type T1 would have all of the components of T0 plus the new components. The components from T0 would be referencable as a group, perhaps as O1.others (presuming O1 : T1). In other words, it would be roughly as though T1 were declared as:

```
type T1 is record
    F1 : integer := E1;
    F2 : float := E2;
    others : T0;
end record;
```

The semantics for calls on the derived subprograms would be by selection of ".others" rather than by conversion, to ensure that fields new to T1 would not be affected by calls on the subprograms inherited from T0. Of course, these derived subprograms could be overridden as usual for a derived type, in which case the new subprograms could access/update the new fields.

Conversion from T1 to T0 would be equivalent to O1.others. Conversion from T0 to T1 would be equivalent to declaring an object of type T1 with the ".others" fields initialized from the T0 object, and with the new T1 fields default-initialized. Conversion between two derivatives of T0 implicitly involves a conversion to their nearest common ancestor type and then to the target type.

A record aggregate for T1 could only use "others" to refer to the T0 components as a group, rather than its usual meaning.

Such a derived type may also add discriminants specified in parentheses immediately after the "with", e.g.:

```
type car is new vehicle with(num_wheels : small_int) record
    pressure : psi_array(1..num_wheels);
    ...
end record;
```

Similar to a record type, it would be useful to be able to add enumerals to an enumeration type, as follows:

```
type E1 is new E0 with (enum1, enum2, ...);
```

The additional enumerals would have to be distinct from the enumerals of E0, and would be assigned position numbers starting at E0'POS(E0'LAST)+1. Conversion from E1 to E0 would perform a constraint check to ensure that the position number of the operand was less than or equal to E0'POS(E0'LAST). Conversion between two derivatives of E0 implicitly involves a conversion to their nearest common ancestor type and then to the target type.

Given this as a mechanism for inheritance, it is useful to be able to talk about a (root) type and all types derived from it. All such types will have some definition for the derivable subprograms of the root type, either by inheriting that of their parent, or by explicitly overriding it with a new definition. Note, however, that we don't want to slow down uses of specific types when known at compile time.

Therefore, it makes sense to define a new kind of type, a polymorphic type, which represents a root type and all of its derivatives. Here is a proposed syntax:

```
type T_Star is all T0;
```

This declares a new type, T_Star which is itself derived from T0, but with a special definition for each of

the derived subprograms. Any value of type T_Star retains an identification of some non-polymorphic type from which it came. When a derived subprogram is called on a value of type T_Star, the retained non-polymorphic type determines which particular definition of the derivable subprogram is called. All IN and IN-OUT T_Star parameters to the same derivable subprogram are required to have the same retained type, and a CONSTRAINT_ERROR is raised if they don't all match. The retained type of an IN-OUT, OUT, or function results of T_Star type will also match this same retained type upon return.

Roughly, T_Star may be thought of as a derivative of T0 with an extra discriminant, as follows:

```
type T_Star is new T0 with (Retained_Type : Type := T0) record
    case Retained_Type is
    when T0 => null;
    when T1 =>  <T1's fields>;
    when T2 =>  <T2's fields>;
            ...
    end case;
end record;
```

Any derivable subprogram with no IN or IN-OUT parameters of T_Star type will be treated as implicitly overloaded on all non-polymorphic types derived from T0, and hence must be resolvable by context to the particular derived subprogram.

Any type derived from T0 is implicitly convertible to type T_Star. Explicit conversion from T_Star to any type derived from T0 is equivalent to converting first to the retained type, and then to the nearest common ancestor, and then finally to the target type.

Objects of type T_Star are like discriminated records, and may be either constrained or unconstrained. Constrained objects only hold values of a single retained type. Unconstrained objects can hold values of any type derived from T0. The maximum size of such values is not known at compile-time, and so generally such objects will be implemented with a hidden pointer. The 'CONSTRAINED attribute may be used to determine whether a parameter of type T_Star is constrained or unconstrained. Local objects of type T_Star are by default unconstrained, though they may be declared constrained as follows:

```
O_param : T_Star(param.retained_type);  -- object with same
        -- retained type as a parameter
```

This is presuming that the implicit discriminant is actually accessible by the name "retained_type" or equivalent.

Here is an example of use, illustrating a heterogenous implicitly-linked list:

```
type Root is record null; end record;
procedure Print_Item(R : Root) is begin null; end Print;
procedure Print_Rest(R : Root) is begin null; end Print;

type Root_Star is all Root;
procedure Print_List(R : Root_Star) is
begin
        Print_Item(R);  -- Print first element of list
        Print_Rest(R);  -- Print rest of list (recursively)
end Print_List;
```

```
type List_Root is Root with record
        Next : Root_Star;  -- Recursive data structure
                   -- Pointer implicitly required
end record;
procedure Print_Rest(L : List_Root) is
begin
        Print_List(L.next);  -- Print rest of list, recursively
end Print_Rest;

type Some_Record is new List_Root with record
        Name : String(1..10);
        . . .
end record;
procedure Print_Item(S : Some_Record) is
begin
        Put(S.name & ' ');  -- Print this item of list
end Print_Item;
```

Note that by judicious definition of derivable subprograms, all explicit case statements may be removed from code, and additional types may be derived from List_Root each with its own definition of Print_Item, with the right one called by
Print_List automatically.

SUBPROGRAM TYPES AND VARIABLES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3

PROBLEM:

Subprogram types and variables have been useful in other languages. Their inclusion in Ada would greatly increase its usability for object-oriented programming, numerical analysis, dynamic configuration of large systems, etc. Generic formal subprograms are not an adequate substitute.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

There are at least two issues that any solution must address. First, there is the problem of dangling references (what happens when a subprogram (and its statically enclosing environment) ceases to exist, but there are variables that may refer to it). And second, how are parameter subtype constraint checks made on indirect calls through a procedure variable. For example, can a procedure with a single Integer parameter be used as a value of a procedure type whose profile specifies a single Natural parameter? And if so, how are the proper parameter constraint checks made on calls to variables of this type? We propose the following solution:

A subprogram type could be declared using a syntax similar to that for task types by including the keyword TYPE in a subprogram specification:

```
procedure type P (X : Integer := 17);
function type F (X : Integer) return Boolean;
```

The predefined operators and operations for a subprogram type are those of a private type (e.g. equality, assignment, qualification) and the operation of calling. The literals of a subprogram type are all subprograms whose parameter profiles conform to that of the type declaration and whose lifetime is no shorter than that of the type(i.e. there exist no task bodies, subprogram bodies, generic specifications or bodies, or block statements that enclose the subprogram declaration but not the subprogram type declaration). These rules guarantee statically that there will be no dangling references.

All subprogram types also have a literal NULL which is the default initial value for all objects of the type. Attempts to call a subprogram variable with the value NULL will raise Program_Error (as will a call to a

variable whose value is that of a subprogram whose body has not yet been elaborated).

The parameter profile of a subprogram is said to conform to that of a subprogram type declaration if they have the same number of arguments and the mode and base type of corresponding arguments are the same.

When a subprogram name is used as a literal of a subprogram type, a check is make that the constraint on each of the parameter subtypes matches that of the corresponding parameter subtype in the specification of the subprogram type. Constraint_Error is raised if the check fails. This will guarantee that constraint checks can be done at the call site using the parameter subtypes of the subprogram type declaration.

```
procedure type P (X : Natural);
function type F return Integer;

function F1 return Integer is ...;
function F2 return Integer is ...;

procedure P1 (X : Natural) ...;
procedure P2 (X : Integer) ...;
procedure P3 (X : Boolean) ...;

Procedure example is
        X : array (1..4) of P;
        Flag: Boolean;
        procedure P4 (X : Natural) is ...;
begin
        X(1) := P4; -- Illegal, lifetime of P4 is shorter than that of type P.

        X(2) := P3; -- Illegal, parameter profile of P3 does not match that of P.

        X(3) := P2; -- Legal, but raises constraint error since the constraints on the parameter
                       subtypes of P2 do not match those of P.

        X(4) := P1; -- Legal, no constraint error.

        X(4) (17);  --  A call to the current value of X(4),
                                        -- i.e., P1.

        Flag := F1 + F2; --Illegal, ambiguous use of "="
end Example;
```

Remaining issues that seem straightforward, but need to be resolved, include interactions with subprogram renames, parameter passing rules for parameters of subprograms types, subtypes and derived types.

PROCEDURE VARIABLES

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
FAX: +46 758 15133

ANSI/MIL-STD-1815A REFERENCE: 3, 6

PROBLEM:

The language lacks the ability to select actions depending on state, except through the use of case statements. Using procedure and function variables in the way defined by RTL/2 would be a simple, elegant, efficient and perfectly safe method.

In our experience, procedure variables came in very handy for instance when you write dispatchers for message handlers.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

State encoding through enumeration types. The resulting case statements are normally a lot less efficient than the resulting indirect subroutine call that procedure variables would mean.

POSSIBLE SOLUTIONS:

Make it possible to declare variables like this:

```
My_Proc : procedure (Left : in Integer; Right : out Natural);
My_Func : function (Input : in String) return Boolean;
```

Then, given the declarations

```
procedure A (First : in Integer; Second : out Natural);
function B (Data : in String) return Boolean;
X: Natural;
Y: Boolean;
```

this would be legal:

```
My_Proc := A;
My_Func := B;
```

```
My_Proc (Left => 123, Right => X);
Y := My_Func(Input => "Marvelous!");
```

It would also be legal to assign an entry name to a procedure variable, if the parameter profiles match.

Note that if procedure/function variables are included as record types; this can provide a cheap form of object declaration; especially if a <> default value convention is chosen so that the default value for the procedure/function variable is take as the procedure by the same name visible at the point of declaration of the record type. This would also require a self function, that always referred to the record instance that contained the procedure/function variable that was called.

A case where this would have been very useful for us is in the area of interprogram communication, where one needs to be able to specify what is to be done with an incoming message. The only way to do this now is through generics, which on most architectures forces code duplication.

INTELLIGENT STRONG TYPING

DATE: October 30, 1989

NAME: David W. Ketchum

ADDRESS: 108 Halstead Ave
Owego, NY 13827

TELEPHONE: (607) 687-5026

ANSI/MIL-STD-1815A REFERENCE: 3, 4.5.5, 4.6, 4.10, 5.2, 6, 14

PROBLEM:

Ada's typing is certainly strong in the sense of being rigid. However, the very rigidity often causes it to interfere with rather than assist in clearly stating a program's goals.

For example, if L, A, and V are declared with types LENGTH, AREA, and VOLUME, addition and subtraction will be fully protected but all reasonable use of multiplication or division will require explicit conversions (or laborious overloading); misuse of the same conversions will permit unreasonable operations while some unreasonable operations (e.g., A:=A*A) will be permitted without conversion. However, if L, A, and V have identical types Ada's typing cannot offer any protection.

IMPORTANCE: IMPORTANT

Presumably the dearth of Ada 9X revision requests on this topic is due to expected futility in trying for improvement, rather that lack of recognition that problems exist. Need for intelligent language support for interaction of units-of-measure data has been discussed many times. The three listed references describe a simplified implementation considered justifiable from savings on a single project, Pascal-oriented details, and Ada-oriented details:

1. G. Baldwin, Implementation of Physical Units, SIGPLAN Notices 22, 8 (1987), 45-50.

2. A. Dreiheller, et al., Programming Pascal with Physical Units, SIGPKAN Notices 21, 12 (1986), 114-123.

3. N. H. Gehani, Ada's Derived Types and Units of Measure. Software-Practice and experience 15 (1985), 555-569.

CURRENT WORKAROUNDS:

1. Where its rigidity interferes, Ada's typing gets disabled.

2. Attributes that cannot be described via Ada typing get documented in commentary, if at all, and related scaling conversions are calculated manually.

POSSIBLE SOLUTIONS:

Dreiheller and Gehani, together, provide a base for constructing a solution. Following text attempts to provide added insight (but without claiming to be a complete design): Here new syntax permits more completely describing attributes of data in Ada declarations, and new semantics rules govern use of such data. However, existing semantics remain undisturbed for code which does not involve variables based on the new syntax. Proposed changes are keyed to current LRM content:

3.1 Add, as a type of declaration: units_declaration

3.2 Add, as content for object_declaration and number_declaration: [units_constraint] This lets simple variables specify units attributes and lets named constants have both numeric value and units attributes (e.g., units RADIANS := 3.14159)

3.3.2 Add, as a type of constraint: units_constraint

A units constraint may be applied to a type mark that already imposes a units constraint--and the effect is the combined effects of the two constraints.

3.3.3 Add, as an attribute: T'UNITS. This returns the units attribute of T, formatted as a character string which could legally be incorporated in Ada source, and ordered in the order that the various base units were declared. NULL is returned if T has no units attribute (whether or not units are permitted for type T).
Among the uses for T'UNITS are to make units attributes accessible to output routines and to make attribute requirements accessible to input routines.

3.5.4 Add, as content for integer_type_definition; [units_constraint]

3.5.7 Add, as content for floating_point_constraint: [units_constraint]

3.5.9 Add, as content for fixed_point_constraint: [units_constraint]

3.10 Add a new section:

3.10 Units Declarations and Constraints

A units declaration assigns a name to a units constraint. A units constraint identifies a collection of units-of-measure attributes.

units_declaration ::=identifier_list : units_constraint;

units_constraint :: = units units_expression

---> No time to complete the formal syntax today, so requirements for units declarations and constraints will be substituted:

1. The basic set of units declarations belong in package STANDARD. Systems International (SI) is a good starting point for determining package STANDARD content, although Ada may wish to omit some and add some (e.g., English units defined in terms of SI base units).

2. A base declaration, such As A for Ampere, m for meter, or s for second, needs a null units attribute and has an implier 1 or 1.0) as scale and an implied 1 as exponent.

It seems appropriate to disable the basic Ada83 type mechanism for units data because it is overly restrictive. and for simple programs the basic units services which guarantee that length, current, area, etc. don't trip over each other should be sufficient. For more complex programs private base dimensions can be added at the point where data originates and accounted for where the data is used.

3. Users must be able to declare new case units, as well as coding secondary declarations involving new scaling and various combinations of previous declarations.

4. While the SI standard prefixes include m for milli and M for Mega, Ada's inability to distinguish case in identifiers bites us once more.

5. Scales must be exact, to minimize losses during conversion. Perhaps the answer is that scales; must be either integers or ratios of integers.

6. Dimensionality (volume = length **3, length = volume **1/3)) also must be exact.

7. Most conversions of scale involve simple multiplications or divisions, but some require invocation of conversion functions (Fahrenheit vs Celsius; decibels; etc.).

8. Sample declarations:

```
m : units NULL; --base unit of length
cm : units (1/100) * m;
liter : units 1000 * cm**3;
inch : units (254/100 * cm;
foot : units 12 * inch;
miles : units 5280 * foot;
s : units NULL; --base unit of time
min : units 60 * s;
hour : units 60 * min;
hour2 : units 3600 * s;
vel1 : units miles/hour
vel2 : units miles * hour**-1,
velc1 : constant units miles/hour :=1;
velc2 : constant units miles/hour :=1.0;
```

Note that hour and hour2 have identical attributes--fact that min was an intermediate step for hour is promptly forgotten. Likewise, vel1 and vel2 are identical functions of seconds and meters with an appropriate scale factor.

velc1 and velc2 are number declarations, not units declarations--they are shown to make the point that vel1 or vel2 may be used as synonymous with either, provided context resolves the question of whether an integer or real is required.

In the code fragment below myalttype and mytimtype are dummy units that are indirectly attached to myaltval, mytimval, and myclimb to ensure that these variables have the desired relationship. Even if calculation of myclimb were much more complex that the simple assignment statement shown, myalttype and mytimtype would have to be propagated correctly to compile successfully. Here we have the same effect on addition as a basic Ada type. We have stronger protection where multiplication or division are involved--units

validation doesn't stop such calculations, but operand units become attributes of the result.

```
myalttype : units NULL:  --a private unit
myalt : units foot  myalttype:
mytimtype : units Null; -- another private unit
mytim : units sec * mytimtype;
myaltval : FLOAT units myalt;
mytimval : FLOAT units mytim;
myclimb : FLOAT units myalt / mytim;
myclimb := myaltval / mytimval;
```

4.5.5 If either or both operands of a multiplying operator have units attributes, then operand type validation is concerned only with the major classes (integer, float, and fixed) and the result has units attributes derived as a function of the attributes of the operands.

4.6 Type conversions involving units attributes require that the type_mark and expression have identical units attributes, other than scaling. If the attributes are identical except for scaling, the expression's numeric value will be scaled accordingly.

4.10 Adjust universal expressions consistent with 4.4.5

5.2 For assignment involving units data, assume an implied type conversion per 4.6

6. As with assignment, units attributes must exactly match.

But consider a function that calculates square root. It cares nothing about the exact units attributes involved, but needs to specify that the attributes of the input parameter be the square of the attributes of the result value.

Then consider a rate smoothing function. It accepts as input Xs and seconds and returns Xs per second. The declaration should indicate the required relationships among the parameter units attributes, working equally well within those requirements for X=feet, X=pounds, or X=liters.

14. Interactive terminal I/O, at least, deserves units support (e.g., via T'UNITS). For output this can be used to attach a description to printed numbers; for input this can be used to specify required units--if a length were required the operator could supply any kind of length and have intelligent scaling applied.

ELABORATION RULES IN THE LANGUAGE THAT IMPACT IMPLEMENTATIONS

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.1, 3.8, 3.9

PROBLEM:

Because of the elaboration rules in Ada, the generated code is far bigger, i.e., less efficient, than for other languages. There is no control for the applications writer to "tame" the elaboration beast. Further, the elaboration rules often alter a dereference such that it accesses a control block and not the item. This interferes with the ability to interface with hardware devices--it is implementation dependent on the number of dereferences required to reach the item.

IMPORTANCE: IMPORTANT

One of the major goals for improving the language for the benefit of realtime embedded applications.

CURRENT WORKAROUNDS:

Avoiding unsuspected elaboration through strict programming standards that (a) allows only minimal package references by WITH, shallow nesting levels, and static structures, (b) substitutes assembler language interfaces, and (c) supplies "special controls" to hide the true nature of the object being referenced.

POSSIBLE SOLUTION:

1. delete #4 in 3.8

2. specifically state that pre-elaboration, where recognized, should be supported. Features/attributes may have to be added to the language.

3. allow constant and literal items to be pre-elaborated or generated at compile time and not on every module reference. Only access rights must be generated, the entire structure of the object does not have to be created. It would be easy to incorporate the distinction in a constant library unit.

4. allow elaboration to occur only on a dereference and not on every Ada LRM declaration. For example, elaboration calls would be generated for such constructs as initialization code in a package and dynamically generated structures, but not for constant or static structures. The compilation system should be able to rely on having a [multiple] symbol table approach for its implementation of elaboration rules.

5. fix the elaboration of access types, which are not exactly subtypes, but are of the same type with a different value, i.e. address. This eliminates the necessity for generating intermediate objects rather than access to the object itself.

6. the elaboration of a formal part should not necessarily have to generate code.

7. specifically state that exceptions need not be raised where the compiler can recognize an erroneous situation and handle the code generation accordingly. Other languages do not have to have provisions for overcoming elaboration errors.

8. program errors should generate error messages at compile time and not merely generate cases for raising exceptions later, such as at runtime. Elaborations for dynamic declarations should only occur at runtime.

ALLOW ANONYMOUS ARRAY AND RECORD TYPES

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 3.2, 3.7

PROBLEM:

The array and record constructs are treated differently from other type constructors, in that they are allowed neither in an object declaration nor in a component of an array or record. Further, the fields of a record are treated differently from object declarations, since they cannot be declared as constrained array definitions. This nonorthogonality leads to clumsy definitions of complex records, extraneous type definitions and hard to read programs. By violating the "law of least astonishment", this restriction makes the language harder to learn.

IMPORTANCE: ESSENTIAL

Regularity is compromised without this, and the design goals of Steelman and of 1.3(3) are violated:

> Concern for the human programmer was also stressed during the design. ... underlying concepts integrated in a consistent and systematic way. ... language constructs that correspond intuitively to what the users will normally expect.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Allow a record definition anywhere that a record type is allowed. Treat such a record definition as an anonymous type, e.g., given

```
A: record
       A1: integer;
       A2: float;
```

```
                end record;
        B:  record
                        A1:  integer;
                        A2:  float;
                end record;
```

A and B would have different types, making A := B invalid, but A.A1 := B.A1 would be valid.

Allow the components of a record to be any type or subtype that is allowed in an object declaration. Treat any array or record specification as an anonymous type.

```
        A:  record
                        A1:     integer;
                        A2:     float;
                        A3:     array (1.10) of integer;
                end record;
        B:  record
                        A1:     integer;
                        A2:     float;
                        A3:     array (1.10) of integer;
                end record
```

A.A3 and B.A3 would have different types, making A.A3 := B.A3 invalid, but A.A3(5) := B.A3(5) would be valid.

Add an additional alternative to the definition of object_declaration in 3.2(9):

```
| identifier_list : [CONSTANT] type_definition [:=expression];
```

Replace

```
component_subtype_indication
```

with

```
component_subtype_indication | component_type_definition
```

in 3.6(2), in suitable metasyntactic brackets.

Add an alternative to the definition of component_subtype_indication in 3.7(2):

```
| component_type_definition
```

with appropriate changes to the text.

ELABORATION OVERHEAD TOO COSTLY

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 3.2.1, 1.4

PROBLEM:

Perhaps the single biggest drawback to Ada compilers for embedded targets is the cost of elaboration. Resources, either memory or processing cycles, expended for elaboration take away from the resources that would otherwise be available to implement the application. We and other people engaged in applications for embedded computers have spent several years attempting to make Ada compilers for our targets suitable for our applications. The Reference Manual, though, takes the singularly unhelpful position that elaboration of objects occurs at execution time. This position is enunciated in the definition of elaboration provided in the glossary. While we realize that the glossary is not a formal part of the standard, in this case it is an accurate reflection of the intent stated in much more obscure language in section 3.2.1. Yet, there is no clearly apparent benefit to deferring elaboration to runtime when it can sensibly be done either at compilation time or at link time.

Dealing with the ramifications of runtime elaboration has cost us and many others substantial amounts of time and money. Compiler vendors we have dealt with seem to have uniformly implemented elaboration of all data at run time -- at least initially. After the needs of our applications have become known, then implementors tend to accept the need for elaboration prior to runtime. However, to reach that point takes time and effort on both the part of the implementor and their customers. Runtime elaboration has significantly held back the acceptance and utility of the language for embedded computer applications.

IMPORTANCE:

Dealing with this issue is perhaps the most important contribution the 9X committee can make.

CURRENT WORKAROUNDS:

Unnecessary expenditure of significant amounts of effort to explain to compiler vendors what needs to be done and to convince them that elaboration prior to runtime is not at odds with the intent of the language.

POSSIBLE SOLUTIONS:

Modify section 3.2.1 to state that elaboration should take place at the earliest possible time. (We realize that not all data can be elaborated prior to execution.) Modify the glossary to be in accordance with this new idea.

Since elaboration has turned out to be such a significant problem with the language, and will continue to be at least a potentially significant problem, elaboration (and in fact the entire execution model) should be discussed in the Language Summary section.

THE MEANING OF CONSTANTS IN ADA

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.2.1

PROBLEM:

In Ada, CONSTANT is not necessarily static, because the programmer can use a function to compute a constant value. It seems that such a loose definition of CONSTANT greatly hampers the ability of the compilation system to recognize static data and to efficiently generate code. CONSTANT should not only mean that the programmer is not allowed to assign a value to it, but also that it is static. We should be able to have true static constants that can be computed at compile time and stored in read-only memory for embedded applications. It makes no sense to "elaborate" such static or literal constructs on every entry. No loss of capability is lost to a programmer by requiring that a constant object be static. The benefit would be great.

Note #21 in the LRM reminds the implementor to treat a constant as nonstatic. This LRM wording for CONSTANT is another motivation for the implementor to re-elaborate every object. At a minimum for the language change, the only expressions that should be allowable would be static or for a discriminant. To add a restriction for a constant declaration to be static does not limit the user as discriminated and constrained items are readily available. Other ways exist in the language to prohibit an assignment to objects, e.g., limited private. The compiler should be able to recognize the difference between static and pseudo-dynamic constants, like a static function call shown in the example, and produce desired efficient code. The vendor never provides the capability out of fear of validation problems over semantics in 3.2.1 and note #21.

IMPORTANCE: IMPORTANT

Lack of ability for the compilation system to truly recognize static data leads to a large percentage of the overhead in the applications.

CURRENT WORKAROUNDS:

None without "skirting" the limits of the language.

POSSIBLE SOLUTIONS:

Add semantics to restrict Constants to static objects.

IMPLICIT CODE/ACTION GENERATION

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.2.1

PROBLEM:

It is unsafe program code generation practice to implicitly take action that has not been specified by the programmer or that is not recognizable in the source code. In the case of an elaborated object initialization, code should not be implicitly computed, e.g., type access set to NULL, rather it should remain undefined, incomplete. The user should be required to provide the default in the subsequent type declaration, i.e., set it to null. Often, programmers are surprised by the effect of elaboration on their program performance, e.g., long elaboration times and excess code generated. The maintenance problem will exist for such implicit actions.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

In programming standards, control initialization and do not allow access types to default. Specify defaults in the type preset. Require the programmer to use minimum number of WITHs and keep the initialization sequences controlled.

POSSIBLE SOLUTIONS:

In the semantics, require access default value in the place of a NULL to be implicitly created. When no value is given, the result should be readily identifiable but not NULL, something like the IEEE NAN for floating point. Require the programmer to write explicit elaboration control or restate the semantics to remove elaboration ambiguous cases.

RUNTIME CONSTANTS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133(fax)

ANSI/MIL-STD-1815A REFERENCE: 3.2.1

PROBLEM:

It would be very useful to have a mechanism that would allow us to declare constants that have no value in the source code, but that are bound to values through some vendor specific method after compilation and linking. The purpose would be to allow tuning and integration-time parametrization to take place without tearing down and recompiling code.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Coding of special packages that achieve the same effect by reading the values from the file system.

POSSIBLE SOLUTIONS:

THE LRM DEFINITION FOR CONSTANTS AND ELABORATION UNNECESSARILY DRIVES THE IMPLEMENTATIONS

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.2.1 p 3-5

PROBLEM:

Compiler implementors faithfully implement the steps in 3.2.1 for every object whether the semantic rule is necessary or even testable, i.e., visible that the compiler implements the process in precisely the manner described in the LRM. Therefore, static objects are recreated on every elaboration and there is no way just to reference the existing object. On searching the LRM for the basis of an elaboration requirement for every construct on each occasion, three main sources exist in the language in spite of later "optimization" wordings.

Two sources, in 3.2.1 (c) and (d) and again in 3.2.2 (a) and (b), direct the implementation to elaborate everything. A specific implementation of this elaboration method is not testable by any verification method in an ACVC test as long as the end results of the declarations are achieved. If the value/size/state is known at compile time, then only the symbol table type information needs to be provided.

The LRM should state the elaboration rules in BNF (Backus Naur Form) and not drive the implementation by semantic rules. Para 10.6 allows optimization of elaboration, but few implementations provide the capability. The main fear is validation suite and the complexity of the language rules. Such semantic complexity of the LRM is feared to contain some remote permutation of code sequences that could generate a compiler elaboration error. Therefore, the vendors play it safe by providing little optimization even for pre-elaborated/known constructs.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

No vendor implements the optimization recommendations due to this LRM definition.

POSSIBLE SOLUTIONS:

Reword page 3-5 based upon principles of inheritance rather than explicit wording that tends to proscribe an implementation, e.g., "the object is created", "any initial value is assigned". That wording is totally unnecessary. The object "could" have already been created and initialized, then the requirement would be to determine the effect of the declarations and references to the [existing] objects.

DEFAULT INITIALIZATION VALUES FOR ALL TYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.2.1(9-13)

PROBLEM:

Uninitialized data is unsafe. Data that is auto-initialized is guaranteed safe, bur the only auto-initializing type provided by the language at present is the record type:

```
type Safe_Record is
        record
                Field 1 : Natural := 0;
                Field 2 : Boolean := False;
        end record;
```

Since all other types are not auto-initializing, they are inherently more dangerous than record types.

In addition, even if a record is auto-initialized, there is no way to find this out (there is no attribute for initialization)

IMPORTANCE: ESSENTIAL

This is not just a matter of aesthetics (although it is admittedly odd that only records can be auto-initializing): Ada is intended to be robust and non-initialized types permit non-initialized object declarations, which are dangerous.

CURRENT WORKAROUNDS:

Some programmers are meticulous enough about initialization that they add an initialization field to their type declarations and check this field before proceeding with operations (raising an exception of the assertion that the object is initialized fails). Unfortunately, this uses additional space for each instance of the type; it also makes type declarations more complicated (consider a simple array type that now becomes a record with an initialized field and the array type as two components).

POSSIBLE SOLUTIONS:

Allow initial values to be specified for all types, not just records. Example syntax might be:

```
type Foo  is new Integer range 1..10 :=9;
type Blip is array (1..100) of Integer := (others => 1);
type Blat is access Foo := null;

Slump : Foo; -- Already had value 9.
```

As with current auto-initialization of record types, explicit initialization should override default value(s):

```
Slump_2 : Foo := 3; -- Has constant value 3.
```

Allow constants of initialized types to be defined without assigning the constant:

```
Slump_2 : constant Foo; -- Has a constant value 9.
```

Add new language-defined attribute called 'INITIALIZED. This should work transitively for types with subcomponents (as 'CONSTRAINED does now).

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will re-compile successfully and will behave identically during execution except for possible small changes in execution speed.

DISCRIMINANT CONTROL

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.3, 3.7.1, 3.7.2

PROBLEM:

It is fine to have discriminated records. However, programming complexity arises from the language definition that is not in the direction of more reliable, easily maintained software. First, the user cannot control where the discriminant is stored. Often, for embedded systems the discriminant may be (a) internal to the record (a component) or (b) external and at a hardware specific memory location. Next, a discriminant should not be allowed to "rewrite" code, e.g., alter the number/type of parameters, array selectors, a slice, or a variable.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Programming standards, assembly language, merge with private and generics that further hide meaning.

POSSIBLE SOLUTIONS:

1. Provide means to control where the discriminant is to be placed and add semantics to the language to show a difference for an external and internal discriminated object. For example,

```
TYPE THING (N:  INTEGER) IS
        RECORD
                N;  --Showing internal storage for N
                Other:  array (1..N) of real;
        END RECORD;
```

as well as allow a compound type statement for externally stored discriminant values:

```
type thing is
        begin
        N: integer:                 --force the discriminant outside the
                                    --record and may be assigned a specific
                                    --memory location by a USE AT clause

        record thing (N);           --may not follow N or even be near
```

```
                                        --the definition of record (N) may not
                                        --be necessary
        other: array (1..N) of real;
      end record;
    end thing;
```

2. Control syntax for where the discriminant can be placed. In particular disallow discriminants from appearing in Return expressions.

3. It would be wise to use either a special symbol to identify discriminants, e.g., %N, or an attribute of the object, e.g., '(N) or .(N). Such an indication keeps discriminated records from looking like an array or a function. With a readily identifiable and controllable method for discriminants, applications code maintainers would not confuse a discriminant with a formal parameter, an array selector, a slice, or a variable.

4. Discriminants should not be allowed to create other side-effects, e.g., change calling/return sequences.

DEFAULT INITIAL VALUES FOR SIMPLE TYPES

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
FAX: +46 758 15133

ANSI/MIL-STD-1815A REFERENCE: 3.3

PROBLEM:

If a type is to have an initial value, it must be a record type. This is an irritating asymmetry of the language; simple types should have the same facility.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Declare encapsulating records.

POSSIBLE SOLUTIONS:

EXTENSION OF INITIALIZATION CLAUSES TO SCALAR TYPES

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 3.3

PROBLEM:

Initialization is permitted in many type declarations, and is in fact implied in access type declarations. It is not, however, permitted in scalar type declarations. If one wishes to export a type which is a key implemented as an integer, with a required initial value which assures it will open no locks until a privilege has been granted, it is necessary to bracket the integer in a record. This lack of uniformity is unnatural.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

```
type Key is record
        Content: integer := 0;
end key;
```

POSSIBLE SOLUTIONS:

Allow initialization clauses on scalar type declarations:

```
type Key is new integer := 0;
```

Such initializations could also be used to create such types as counters, which are conceptually initialized to zero. Initialization to unnatural values (such as integer'first) could be used to show up errors.

IMPLICATION THAT VALUES CAN BE ASSIGNED TO TYPES

DATE: May 12, 1989

NAME: George A. Buchanan

ADDRESS: IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

TELEPHONE: (301) 731-8894 ext. 2063

ANSI/MIL-STD-1815A REFERENCE: 3.3(6)

PROBLEM:

Because of imprecise wording the use of (default initial)values may be misinterpreted. This is particularly illustrated in Section 3.3, Paragraph 6, which could be misinterpreted to mean that types (rather than access objects or record components) can be assigned values.

IMPORTANCE: ADMINISTRATIVE

If this wording is not corrected, then the Ada reference manual will continue to fail to convey what its authors really meant, thus resulting in unnecessary coding errors and compilation errors.

CURRENT WORKAROUNDS:

Hopefully readers of the Ada reference manual will correctly understand what the wording was meant to convey despite the imprecise wording.

POSSIBLE SOLUTIONS:

In Section 3.3, Paragraph 6, if "certain types have default initial values defined for objects of the type" refers to objects of access types, then clearly indicate this reference. Also in Section 3.3, Paragraph 6, if "certain other types have default expressions defined for some or all of their components" refers to the components of record types, then clearly indicate this reference.

OPTIONAL DEFAULT INITIALIZATION FOR ANY USER-DEFINED TYPE

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP:jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 3.3 (6), 3.7 (5)

PROBLEM:

In general, objects of any type, in particular of any scalar type, cannot be initialized by default. They can only be initialized explicitly in the object declaration. Currently, the only objects which can thus be initialized by default (by adding an initial expression to the type definition) to take the same value for all objects of their type are non-limited components of a record type. We require that objects of any type, in particular of any scalar type, can be initialized by default by adding an initialization to the type definition.

IMPORTANCE: IMPORTANT

The current non-uniformity in the language is in conflict with the language design requirements and is likely to cause human errors.

CURRENT WORKAROUNDS:

Initialization in all object declarations with the danger that a changed context might make obscure a change of the value returned by the default expression.

POSSIBLE SOLUTIONS:

In ARM 3.3.1 (2) to allow (in the syntax for type_declaration) an addition " [:= default_expression] " where the expression is of the (sub)type given by the preceding type_definition.

INSUFFICIENT TYPE DESCRIPTOR ACCESS

DATE: July 3, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10 - Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 3.3.1

PROBLEM:

A lot of problems which can be solved once and for all by a general solution, need to be solved time and time again in Ada because the language does not support re-usability to that extent. Good examples are persistent objects, textual and binary sequential I/O of composite types, user interface mgmt., SQL interface. In all these instances a centralized solution is feasible.

The simplest example is textual output. For scalar types, textual output is supported. For array types, one can write a generic parameters. For record types, nothing can be done to prevent the programmer from repeating the record declaration in some other way if one stays within the language.

As a result, people tend to think about preprocessor to avoid this lack of reusability support. Indeed, information duplication is not just more development effort, it is far worse since it almost guarantees maintenance problems: a modification in the information source is likely to be forgotten in the information duplicate.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Extend the language to support runtime types : a package that allows read access to objects that represent the type declarations in the source, similar to a standard interface to debugged symbol tables. Another package then could support runtime typed objects : a limited private type that associates an address with a type descriptor reference. This package could support a new kind of unsafe conversion:

```
type T_DYMANIC is limited private;
generic
 type T_STATIC is limited private; -- Really any type at all.
procedure CONVERT ( STATICALLY_TYPED : in out T_ STATIC:
                    RUNTIME_ TYPED : in out T_DYNAMIC);
```

The basic unsafety of this conversion would be the fact that one creates an access path to a memory

location that could be destroyed, hence a better solution is:

```
type T_DYMANIC is limited private;
--- Address of a dynamic object together with its type descriptor.

generic
 type T_STATIC is limited private; -- Really any type at all.
 type T_REF is access T_STATIC;
procedure CONVERT (STATICALLY_TYPED : in out T_REF;
                   RUNTIME_TYPED : in out T_DYNAMIC);
```

Which only creates an alternative (and typed!) access path to a dynamically created object.

The rest of the package should then support all type-specific operations, use of which will be checked with the runtime type. Hence, a program might find out that a T_DYNAMIC object has a certain record type, he can visit the record components, their names, types, and values (again of type T_DYNAMIC). For some components it might find out that they belong to array types, find the number of indices, their type and the component subtype, he can visit the components, find some of an integer type and get their values in an INTEGER variable, etc...

Obviously a large effort is required to create a neat and maximally safe specification of these packages. Compared to TEXT_IO, I would guess this will take 2-4 times the effort spent, but in contrast to TEXT_IO, there is not way the programmer could implement this package himself.

CONSISTENT SYNTAX FOR TASK TYPE DECLARATIONS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95045-3197

TELEPHONE: (408) 496-3706 (11am--9pm)

ANSI/MIL-STD-1815A REFERENCE: 3.3.1(2), 9.1(3, 8)

PROBLEM:

The syntax for task types is different from the syntax for all other type declarations (including the proposed syntax for subprogram types documented elsewhere in these submissions), making it harder to learn (students cannot leverage off of their understanding of the syntax of other type declarations).

IMPORTANCE: IMPORTANT

Students have been known to complain that they need to memorize a special syntax for task type declarations for no apparent reason; any such obstacle (no matter how trivial) is yet another disincentive to learning the language.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Change the syntax of task type declarations as follows:

```
type Resource is task
        entry Seize;
        entry Release;
end Resource;
```

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful re-compilation of previously-compiled code is not guaranteed (although it is easy to fix).

SUBTYPE INHERITANCE OF THE "=" OPERATOR

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 3.3.2

PROBLEM:

If a subtype is declared, equality tests cannot be performed (simply) on instances of the type unless the "=" operator is renamed. Unfortunately, the operator has to be renamed from the package where the base type was declared. This is not intuitively obvious to the Ada non-expert. (This applies to a lesser extent to the other operators defined for scalar types).

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Lots of renames

POSSIBLE SOLUTIONS:

Make the "=" operator automatically visible after the subtype has been declared.

PROVISION OF A SUPERTYPE CAPABILITY

DATE: October 3, 1989

NAME: Eric Kiem

ADDRESS: C/- 3201 Meadowood Drive
Midwest City, OK 73110

TELEPHONE: 734-2457

ANSI/MIL-STD-1815A REFERENCE: 3.3.2, 3.5, 3.5.5

PROBLEM:

The Ada language provides for an existing type declaration to be split into a series of subordinate declarations, via the subtype declaration. A Supertype declaration would permit a series of subordinate discrete types to be integrated to produce a higher, aggregate type declaration, with a resultant improvement in reusability.

For example, with a base declaration of:

```
type Days_of_The_Week is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);          (1)
```

We can extract subranges such as:

```
subtype Week_Days is Mon .. Fri;                                       (2)
        and
subtype Week Ends is Sat .. Sun;                                       (3)
```

A supertype would provide the converse capability. For example, if we have base declarations of:

```
type Week_Days is (Mon, Tue, Wed, Thu, Fri);                           (4)
        and
type Week_Ends is (Sat, Sun);                                          (5)
```

then the aggregate (supertype) would be produced by a declaration of the form of:

```
supertype Days_of_The_Week is (Week_Days,Week_Ends);                   (6)
```

The resultant declaration produces the equivalent of the explicit Days_of_The_Week declaration (1), but it is based only upon the original base declarations (4) and (5).

IMPORTANCE: IMPORTANT

Without a supertype capability, otherwise reusable packages may need to be modified to provide visibility to some all-encompassing discrete type declaration. The practice of amending reusable packages degrades the reliability of reusable components, and therefore should be avoided wherever possible.

CURRENT WORKAROUNDS:

Consider that we possess two packages of DATE and PERSON, with partial specifications of:

```
package DATE is

    type Date_Type is private;

    type Field_Type is (Year,Month,Day);

    procedure Assign      (Image : IN      String;
                           Date : IN OUT Date_Type;
                           Field: IN     Field_Type);

    ...

end DATE;

package PERSON is

    type Person_Type is private;

    type Field_Type is (SSN,Name,Address);

    procedure Assign      (Image : IN      String;
                           Person: IN OUT  Person_Type;
                           Field : IN      Field_Type);

    ...

end PERSON;
```

Within each package, the Field_Type is used to control the target of the operations exported by each package. Each package is reusable, because each is complete, self-contained, and independent of any specific application.

Now, consider an application which requires that we construct an abstract data type which possesses elements of both Date_Type and Person_Type. For example, we may require a membership package, which identifies the Person, and the Date at which membership commenced. We may declare:

```
with PERSON:
with DATE;

package MEMBER is

    type Member_Type is private;

    type Field_Type is (SSN,Name,Address,Year,Month,Day);

    procedure Assign(Image  : IN      String;
        Member : IN OUT  Member_Type);
                           Field  : IN      Field_Type);
```

```
        ...

    end MEMBER;
```

Presumably, the MEMBER package will wish to merely re-direct each operation to the appropriate subordinate package, according to the value of the Field parameter. However, we are faced with determining the most appropriate implementation of MEMBER.Field_Type. A number of alternatives exist:

a. an aggregate Field_Type can be declared in a common package, from which the DATE and PERSON packages declare appropriate subtypes.

For example:

```
package AGGREGATE is

    type Field_Type is (SSN,Name,Address,Year,Month,Day);

end AGGREGATE;

with AGGREGATE;
package DATE is
    ...
    subtype Field_Type is AGGREGATE.Year .. AGGREGATE.DAy;
    ...
end DATE;

with AGGREGATE;
package PERSON is
    ...
    subtype Field_Type is AGGREGATE.SSN .. AGGREGATE.Address;
    ...
end PERSON;
```

The Field_Type within MEMBER then simply becomes a subtype of AGGREGATE.Field_Type;

b. MEMBER can declare an aggregate enumerated type, and can internally "map" the Field_Type to the corresponding subordinate enumeration literal, for example:

```
begin
    Case F is
        ...
        When MEMBER.Month => return DATE.Month;
        ...
    end case;
end Date_Field_Corresponding_To;
```

A further possible implementation of this alternative is to calculate the correspondences based on ordinal values and offsets within the base and aggregate declarations. This may be performed explicitly or by instantiation of an appropriate generic.

Both alternatives have undesirable implications:

Alternative a. introduces an application dependency which degrades the reusability of either package. Furthermore, a recompilation is required, which may destroy existing references to those library units, and require drastic recompilation of entire libraries.

Alternative b. is preferable, because the reusability is retained and recompilation is avoided, but it suffers from the manual requirement to map the enumeration literals within the aggregate Field Type to the corresponding literals within the base declarations, with the possibility of introducing incorrect mappings. In addition, should either base declaration change, for example by the provision of an additional Field literal, corresponding changes will be required within the integrating package and may be overlooked.

POSSIBLE SOLUTIONS:

The proposed solution is to provide the mechanism described in alternative b. as a compiler function. That is, the compiler has knowledge of the necessary ordinal values and offsets within the base declarations and the aggregate declarations to automatically effect the necessary mappings. This implementation ensures that the correct mappings are propagated as subordinate base declarations change, and removes the possibility of incorrect mappings arising from manual mappings.

Presumably, the Ada language would be changed to permit a declaration of:

```
supertype_declaration ::=
   supertype identifier is
        (enumeration_type_definition {,enumeration_type_definition})
```

The mappings between the supertype and base type declarations would presumably be accomplished using a modified form of type conversion (in which the appropriate offsets are subtracted, etc) such as:

DATE.Field_Type(MEMBER.Month) yields DATE.Month,

and

MEMBER.Field_Type(DATE.Month) yields MEMBER.Month.

UTILITY OF ATTRIBUTE 'BASE SHOULD BE EXPANDED

DATE: August 17, 1989

NAME: James W. McKelvey

ADDRESS: R & D Associates
P.O. Box 5158
Pasadena, CA 91107

TELEPHONE: (818) 397-7246

ANSI/MIL-STD-1815A REFERENCE: 3.3.3, 9

PROBLEM:

When writing generics, it is often necessary to define types or subtypes similar to generic formal parameters. But this is difficult because generic actual parameters may have range constraints. Example:

```
type In_Integer is range <>;              -- generic formal parameter
subtype In_Type is Integer range -1 .. 5; -- generic actual parameter
```

The actual parameter may be of any integer type, and may have any range constraints. For a given package (say a random number generator) we need a type that is similar to In_Integer, but that allows a full range of values; i.e., that has no constraints. We could try:

```
subtype X is In_Integer range - In_Integer'Last .. In_Integer'Last;
```

but this won't do if the range of In_Integer is more restrictive, so try

```
type X is range - In_Integer'Last .. In_Integer'Last;
```

but this won't do because the ranges are non-static, so try

```
type X is new Integer range - In_Integer'Last .. In_Integer'Last;
```

but this won't do because it is non-portable due to uncertainty about what exactly is type Integer on a particular implementation.

The only reliable solution is:

```
type Maximum_Integer is System.Min_Int..System.Max_Int;
subtype X is Maximum_Integer range - In_Integer'Last .. In_Integer'Last;
```

This is OK, as long as the largest integer type of the implementation meets all requirements. If it doesn't we cannot instantiate the generic on this implementation anyway.

But this solution is too conservative. It could be that the largest integer type available on the system is excessively large and inefficient compared with the actual generic parameter.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: As described above.

POSSIBLE SOLUTIONS:

A far better solution could be obtained if the attribute 'Base was made more general. Example:

```
subtype X is In_Integer'Base range - In_Integer'Last .. In_Integer'Last;
```

In other words, allow the 'Base attribute generally, so as to access the BASE type of a given type.

TOPIC "=" AS A BASIC OPERATION ?

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3.3.3

PROBLEM:

Very frequently, the sole cause for a 'use'-clause in Ada programs is to obtain direct visibility to the equality operation, which for all but limited types cannot be hidden by a user-provided function definition. These 'use'-clauses are detrimental to code readability and lead to potential overload resolution problems (ambiguity rules).

The write-around of renaming equality locally is ugly and often not applied.

Yet, the rules of the language make it completely obvious that, for non-limited types, equality can bind only to the predefined operation, so that direct visibility of the type declaration and its implicitly declared equality cannot possibly alter the meaning of the operation. (ARM 6.7 (4+5)). For limited types, one could either stay with the current rule of requiring direct visibility, or one could limit the opportunity to declare quality to the same declarative part in which the declaration of the limited type occurs.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

use clauses, renaming; none satisfactory

POSSIBLE SOLUTIONS:

It should be considered whether equality ("=", "/=") should be reclassified to be a basic operation, with a special rule that, for limited types, the definition of "=" provides the definition of this basic operation.

ELIMINATION OF ANONYMOUS ARRAY TYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706[11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.3.5

PROBLEM:

Anonymous Array Types encourage bad programming habits. If this is a desirable goal then we should be encouraging everyone to write in C instead of Ada.

IMPORTANCE: *IMPORTANT*

Anonymous Array Types should never have been allowed in the standard in the first place.

CURRENT WORKAROUNDS:

Good programming discipline, coding standards, code audits, code checking utilities.

POSSIBLE SOLUTIONS:

Eliminate Anonymous Array Types from the language.

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed. On the other hand, only badly-written code will fail to compile, so this is not great loss.

PHYSICAL DATA TYPES

DATE: September 18, 1989

NAME: Jehuda Ziegler

ADDRESS: ITT Avionics
Dept. 73813
390 Washington Ave.
Nutley, N.J. 07110

TELEPHONE: (201) 284-2030

ANSI/MIL-STD-1815A REFERENCE: 3.4, 4.5.3, 4.5.5

PROBLEM:

Ada does not define physical data types with predefined operations as allowed by the physical dimensions that they represent. Currently these quantities can be defined as derived types of integer, real, or floating point types or as private types with a user defined set of allowed operations.

Using derived types implies that all predefined mathematical operations (+, -, *, /, **) between two operands of the same type are allowed and result in an output of the same type. This in general is true for addition/subtraction of physical units (volts+volts => volts), but not for multiplication/division and exponentiation (volts*volts /=> volts). The only way to restrict these operations is to redefine the predefined operation with functions that raise exceptions at run time (this is not as clean as compiler prohibited operations).

Using private types to define physical quantities eliminates these illegal operations cleanly, but eliminates the ability to define subtypes in other packages to limit the range of the defined general physical type for specific applications (such as the range of an instrument or the speed of an airplane).

Multiplication/division of any physical type by any dimensionless scalar type (integer, real, and float) should also be allowed in all situations and provide a result of the same physical type (float(3.0) * volts => volts). Using derived or private data types will require the definitions of these operations separately for each physical type.

The IEEE STD 1076-1987 VHSIC Hardware Description Language (VHDL) has defined physical data types with predefined addition/subtraction, and multiplication/division by scalars.

IMPORTANCE: IMPORTANT

This would enhance the use of Ada for real world applications where operations on physical quantities are important. It would also help Ada perform the functions now being done by special purpose languages such as VHDL for hardware design and ATLAS for ATE applications.

CURRENT WORKAROUNDS:

A user could define all the physical types as derived types or private types as defined above, with the

limitations described above (raising exceptions at run time for derived types, and not allowing user-defined subtypes for private types).

POSSIBLE SOLUTIONS:

The LRM should define physical types with predefined addition/subtraction between operands of the same type, and multiplication/division by scalars. Multiplication/division of physical types should not be allowed by the compiler except as defined by specific application packages (this is better that raising exceptions at run time for illegal operations).

An additional option would be to define a standard package defining all international (SI) physical data types and the allowed set of operations between them.

REFERENCES:

IEEE Std 1076-1987 VHDL Language Reference Manual, sections 3.1.3, 7.2.3, 7.2.4, and 7.2.5.

ADA SUPPORT FOR ANSI/IEEE STD 754

DATE: September 1989

NAME: Randal Leavitt (Canadian AWG #005)

ADDRESS: PRIOR Data Sciences Ltd.
240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 3.4, 3.5.6, 3.5.7, 3.5.8, 4.5.3, 4.5.4, 4.5.5, 4.5.6, 4.5.7, 4.6, 12.1.2, 12.3.3, 13.7.3, 14.3.8, 14.3.10, Appendix C

PROBLEM:

The Ada standard does not specify an interface that can be used to write portable applied mathematics programs. Also the Ada standard model for floating point goes not provide features defined by other widely used floating point standards such as ANSI/IEEE Std 754-1985. These omissions hinder the development of new programs. They also make it difficult to develop reusable mathematical components libraries.

An important consequence of this omission is that basic functions such as square root are not tested during compiler validation. These functions are often critically important for embedded applications, and should be validated. An extensive test suite already exists for ANSI/IEEE Std 754-1985 and it could be added to the Ada validation suite.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

The only method available now is the brute force approach of packaging mathematical functions in some arbitrary manner and using these packages for applications. This simplifies but does solve the portability problem, and often causes the generation of less than optimally efficient code. For example, the code needed for operations such as factoring out the sign, exponent, and mantissa of a floating point number must be either totally non-portable or extremely inefficient.

POSSIBLE SOLUTIONS:

Extend the predefined Ada floating point types to bind Ada with ANSI/IEEE Std 754-1985 by adding a predefined type FLOAT_STD_754. This will provide values such as "not a number" and "infinity" for floating point variables of this type. Some means of expressing these values as literals will be required. Also include mandatory generic Ada interfaces for the important mathematical functions listed in the "Proposal for Standard Mathematical Packages in Ada" by J. Kok, Report NM-R8718, 1987 and require these generic interfaces when instantiated with the FLOAT_STD_754 type to be validated.

IMPROVING DERIVED TYPES

DATE: October 22, 1989

NAME: Arnold Vance

ADDRESS: Afflatus Corp.
112 Hammond Rd.
Belmont, MA 02178

TELEPHONE: (617) 489-4773
E-mail: egg@montreux.ai.mit.edu

ANSI/MIL-STD-1815A REFERENCE: 3.4, 7.4

PROBLEM:

When implementing closely-related packages it is sometimes desirable to derive a private type in one package from a private type in another package and also have access to the representation of the derived type (only within the private part, of course). There is no way to accomplish this.

IMPORTANCE: ESSENTIAL

Derived types should be useful for layered development of packages with private types.

CURRENT WORKAROUNDS:

It is desired to have the following:

```
package pkg1 is
        type t1 is private; -- desired parent type
        ...
private
        ...
        end pkg1;

with pkg1; use pkg1;
package pkg2 is
        type t2 is new t1; --derived type
        ...
private
        --want to be able to access representation of t2 at this point but can't
        ...

        end pkg2;
```

The following is an approximation of what is desired:

```
package pkg1 is
```

```
        type t1 is ...; -- move full type declaration here
        ...
        end pkg1;

with pkg1; use pkg1;
package pkg2 is
        type t2 is private;
        -- declare all subprograms that t2 derives in private part
        ...

private
        type t2 is new t1;
        end pkg2;

package body pkg2 is
        -- implement all declared subprograms using conversion
        ...
        end pkg2;
```

This solution has the unfortunate side effect of exposing the representation of t1 not just to pkg2 but to any user of pkg1. A shell subprogram for each of the derived subprograms must be implemented using conversion techniques.

POSSIBLE SOLUTIONS:

A proposed solution is to extend the existing mechanisms as illustrated in the following reformulation of the first example:

```
Package pkg1 is

        type t1 is private except pkg2;  -- This says that pkg2 may have access
                                         -- to representation of t1 if desired.
                                         -- After 'except' a list of package names
                                         -- is allowed.
                                         -- (could use 'exception' to save keyword)
        ...
private
        ...
        end pkg1;

with pkg1; use pkg1;
package pkg2 is

        type t2 is private new t1;       -- The interpretation is the (hypothetical)
                                         -- union of the two statements:
                                         -- type t2 is private;
                                         -- type t2 is new t1;
                                         -- The representation of t2
                                         -- is made available only in the private part
                                         -- but all derivable subprograms of t1 are
                                         -- implicitly declared here
```

```
        ...

private
        --may access representation of t2 at this point
        --no need for shell subprograms
        ...
        end pkg2;
        with pkg1; use pkg1;
```

If pkg2 was not in the list of package names after 'except', it is an error that should be detected when pkg2 is compiled. It should be noted that the list of packages allows the package designer to tightly control the extent to which the representation of a type is known outside its defining package.

IMPROVED INHERITANCE WITH DERIVED TYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.4, 7.4.2(5), 8.3(17)

PROBLEM:

Inheritance has become one of the standard attributes of modern object-oriented programming languages (such as C++ and Smalltalk-80). Unfortunately, Ada is quite deficient in its support for inheritance (it is based primarily on derived types, and then not particularly well), and this is a valid criticism leveled at the language by critics (and C bigots who, if forced to learn a new language, simply prefer to learn C++). There are currently many proposals to add full-blown inheritance (and other standard object-oriented attributes, such as polymorphism) to Ada; the scope of this revision request is much more modest, intended only to make the derived type mechanisms that already exist work better.

IMPORTANCE: ESSENTIAL

If the lack of modern object-oriented attributes is not addressed in Ada 9X, Ada will almost certainly become the FORTRAN of the '90's.

CURRENT WORKAROUNDS:

Be thankful for what limited object-oriented support is offered by the current language.

POSSIBLE SOLUTIONS:

To start with, consider the following situation:

```
package Some_Package is

    type Some_Type is private;

    function Some_Function  (On_This : in     Some_Type)
                    return Boolean;
    procedure Some_Procedure (On_This : in out Some_Type);
```

```
private
        ...
end Some_Package;

with Some_Package;
package Another_Package is

        type Another_Type is private;

        function Another_Function  (On_This : in     Another_Type)
                                   return Boolean;
        procedure Another_Procedure (On_This : in out Another_Type);

private

        type Another_Type is new Some_Package.Some_Type;

end Another_Class;
```

The way the language is currently defined, the declarations of Some_Function and Some_Procedure in Another_Package hide the implicit declarations of operations by the same names (and with the same parameter profiles) that were inherited by the type derivation of Another_Type. Now, in order to implement Some_Function and Some_Procedure in the body of Another_Package it is necessary to coerce the types back to the base type:

```
package body Another_Package is

        function Some_Function (On_This : Another_Type) return Boolean is
        begin
                return Another_Type
                        (Some_Package.Some_Function
                        (Some_Package.Some_Type (On_This)));
end Some_Function;
procedure Some_Function (On_This : Another_Type) is

                Some_Local : Some_Package.Some_Type :=
                        Some_Package.Some_Type (On_This);
begin
                        Some_Package.Some_Procedure (Some Local);
                        On_This := Another_Type (Some_Local);
end Some_Procedure;

end Another_Package
```

This is completely backwards from the way inheritance is supposed to work. For proper inheritance, only in the case where the programmer wants to explicitly override an inherited operation should the programmer have to do any work at all. Thus, the way the language should work in the above case is that at the point that Another_Type is derived, the operations Some_Function and Some_Procedure defined in Another_Package package spec should be considered already implemented, and, in fact, it should not even

be necessary to create a package body for Another_Package: the programmer is done.[1]

Now, consider the case where the programmer does, in fact, want to override one of the operations.In that case, the programmer would need to create a body for Another_Package, but would only need to fill it in with the operation that is to be overridden:

```
package body Another_Package is

        function Some_Function (On_This : Another_Type) return Boolean is
        begin
                [do something new here]
        end Some_Function;

end Another_Function;
```

Notice that Some_Procedure does not have to be implemented, because it is already implicitly obtained from direct inheritance. Only the operation that the programmer wants to override needs to appear in the package body. This is the way inheritance is supposed to work.

Now consider the case where the programmer wants to directly inherit operations but change their names:

```
package Some_Package is

        type Some_Type is private;

        function Some_Function  (On_This : in Some_Type)
                        return Boolean;
        procedure Some_Procedure (On_This : in out Some_Type);

private
        ...
end Some_Package;

with Some_Package;
package Another_Package is

        type Another_Type is private;

        function My_Function  (On_This : in    Another_Type)
                              return Boolean;
        procedure My_Procedure (On_This : in out Another_Type);

private

        type Another_Type is new Some_Package.Some_Type;
```

---

[1]If leaving the body non-existent is to indeterminate, it would be acceptable to this author to require the programmer to at least compile a null body for the package in order to indicate that it is indeed intended to be implemented entirely by inheritance.

```
end Another_Class;
```

When it comes time to implement the body of Another_Package (which is now required because the operations that are inherited do not exactly match the operations that are declared, because the names are different), it should be possible to do so via simple renaming of the inherited operations:

```
package body Another_Package is

    function My_Function (On_This : in Another_Type) return Boolean
        renames Some_Function;

    function My_Procedure (On_This : in out Another_Type)
        renames Some_Procedure;

end Another_Package;
```

This is very nice because it eliminates the run-time overhead of a call to what would just be a "skin" anyway (of course, if the programmer wants to override the default implementation, that is certainly possible as well). It should also be possible to use this same strategy to implement operations that are the same as the derived operations except for different default values for parameters (as is already possible with renames).[2]

Note: It should be pointed out that nothing above is meant to imply that operations defined in a package spec that do not match any derived operations do not have to be implemented by the programmer: it is certainly true that--if Another_Package defined a brand new operation called Brand_New_Operation that had a parameter profile unlike either of the two inheritable operations defined in Some_Package--this new operation would have to be implemented in the body of Another_Package. This of course consistent with the way inheritance is supposed to work (the programmer can subset, superset, or merely duplicate the operations inherited from the parent).

Assuming all of the above is fixed, the one other gross deficiency in the current standard with respect to its support for inheritance via derived types is that generic subprograms are not inheritable [LRM 3.4 (20)]. This automatic subsetting of inheritable operations to exclude generics is a wretched restriction, since it is exactly the interaction of generics with inheritance properties that shows the most promise in at least emulating polymorphism. Recommended solution: relax this restriction so that generic subprograms are inheritable.

COMPATIBILITY:

Oddly enough, all of the above proposed changes are upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

---

[2]There is no reason why the ability to implement functions and procedures declared in package specs by renaming in the package body should be restricted to derived types; it should be possible to do in any case where the renaming rules are not violated, since the elimination of run-time overhead and skin operations is always desirable.

PROVIDE EXPLICIT SUBPROGRAM DERIVATION

DATE: October 23, 1989

NAME: Allan R. Klumpp

ADDRESS: Jet Propulsion Laboratory
4800 Oak Grove Drive
Mail Stop 301-125L
Pasadena, CAL 91109

TELEPHONE: 818-354-3892
FTS 792-3892
Internet: KLUMPP@JPLGP.JPL.NASA.GOV
Telemail: KLUMPP/J.P.L.

ANSI/MIL-STD-1815A REFERENCE: 3.4.11

ALIWG ACTION: Favorable vote 1987 in Boston, tie vote 1988 in Charleston, W.V.

PROBLEM:

When a subprogram has multiple parameter and/or result types from which multiple new types are subsequently derived, the only subprograms that are derived are those that differ from the original subprogram in exactly one parameter or result type. None of the derived subprograms is usable in the normal manner.

Example

```
PACKAGE VECTOR_MATRIX IS
        TYPE VECTOR IS ARRAY(INTEGER RANGE <>) OF FLOAT;
        TYPE MATRIX IS ARRAY(INTEGER RANGE <>, INTEGER RANGE <>) OF FLOAT;

        FUNCTION "*" (LEFT: VECTOR; RIGHT: MATRIX) RETURN VECTOR;
END VECTOR_MATRIX; -- End Spec

WITH VECTOR_MATRIX; USE VECTOR_MATRIX;
PACKAGE QUATERNION IS
        TYPE QUAT IS NEW VECTOR(0 .. 3);
        TYPE ROT_QUAT IS NEW MATRIX(0 .. 3, 0 .. 3);
END QUATERNION; -- End Spec
```

The product function "*" is derived twice. One derived function takes QUAT and MATRIX parameter types and returns a QUAT type. The other derived function takes VECTOR and ROT_QUAT parameter types and returns a VECTOR type. Neither derived function is usable in the normal manner, i.e., as follows:

```
        QUAT_A, QUAT_B:    QUAT;
        ROT_QUAT_B_A:      ROT_QUAT;
BEGIN
```

```
QUAT_B := QUAT_A * ROT_QUAT_B_A;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Make the subprogram generic with the generic formal parameters being the parent types. Then instantiate the subprogram for the derived types. This workaround is often unsatisfactory because other subprograms call the subprogram in question. In such cases, a second, nongeneric, subprogram must be provided to satisfy the callers.

POSSIBLE SOLUTIONS:

The language should enable the applications program to declare derived subprograms, naming the derived types they use. (For compatibility with existing software, those derived subprograms the language supplies implicitly should continue to be supplied.) Here is a deriving declaration with which an applications program could declare derived subprograms.

```
Deriving_declaration ::=
        subprogram_specification is new subprogram_specification;
```

Using the deriving declaration, the quaternion product function in the example could be declared by:

```
FUNCTION "*" (LEFT: QUAT; RIGHT: ROT_QUAT) RETURN QUAT IS NEW
        FUNCTION "*" (LEFT: VECTOR; RIGHT: MATRIX) RETURN VECTOR;
```

The right-hand subprogram_specification disambiguates the deriving declaration. Disambiguation is necessary because more than one parent subprogram could be implied by the left-hand subprogram_specification and there may be no other way to distinguish among the possible parents.

Additional rationale on disambiguation and possible wording for the Ada 9X LRM are available on request.

SCALAR TYPE DEFAULTS

DATE: July 21, 1989

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of Ada UK.

ADDRESS: IPSYS plc
Marlborough Court
Pickford Street
Macclesfield
Cheshire SK11 6JD
United Kingdom

TELEPHONE: +44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 3.5

PROBLEM:

In some situations it is necessary to ensure that all objects of a particular type have a default initial value. This can only be achieved through the use of a record type definition, in which the components have default values, or through the use of an access type, which results in a default value of null. It is not possible to associate default initial values with other type definitions.

IMPORTANCE: IMPORTANT

To avoid unnecessarily contorted type definitions.

CURRENT WORKAROUNDS:

For some situations the use of record or access types may be a natural solution.

For situations where the properties of a scalar type are required, the obvious workaround is to declare the type as a record thus:

```
type LIKE_scalar is
        record
                VALUE : scalar := default;
        end record;
```

Other workarounds rely on the user of the type to make some explicit initialization, or to call some initializing procedure associated with the type.

POSSIBLE SOLUTIONS:

Allow default expressions for enumeration, integer, and real type definitions and array and derived type definitions thereof.

As an example:

```
type STATUS is (IDLE ,ACTIVE) := IDLE;
type INDEX is new INTEGER := -1;
```

RESTRICT NULL RANGES

DATE: March 20, 1989

NAME: J A Clare (endorsed by Ada UK and Ada-Europe)

ADDRESS: ICL Defence Systems
Eskdale Road
Winnersh
Wokingham
Berkshire RG11 5TT
United Kingdom

TELEPHONE: +44 734 693131

ANSI/MIL-STD-1815A REFERENCE: 3.5, 3.6.1

PROBLEM:

Null ranges R such that R'LAST < R'FIRST-1 (herein referred to as "sub-null" ranges) are somewhat anomalous, particularly when used as index constraints. For arrays with such constraints the LENGTH attribute is not given by the normal formula of LAST - FIRST + 1. This causes significant overheads for the common case of an index constraint with at least one dynamic bound, particularly for a machine with hardware index checking based on lower bound and length.

As it is hard to imagine a good, sensible use of arrays with subnull index constraints, it seems wrong that such an overhead should be imposed on arrays generally.

IMPORTANCE:

CURRENT WORKAROUNDS: Not applicable

POSSIBLE SOLUTIONS:

The most radical solution is to require the evaluation of a subnull range to raise CONSTRAINT ERROR. This would however put overheads on some loop statements, for example.

A less radical solution would be to require the use of sub-null ranges as index constraints raise CONSTRAINT ERROR.

A partial solution would be to redefine the LAST attribute for all null arrays and types to be FIRST-1, to avoid the necessity of passing the upper bound about (e.g. with an array actual parameter where the formal parameter is unconstrained), just for the purpose of evaluating this attribute.

NULL SPECIFICATION FOR NULL RANGES AND RAISING EXCEPTIONS ON NULL RANGE ASSIGNMENT ERRORS

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.5, 4.3.2 #14

PROBLEM:

Ada should have a better notation for indicating a null range, e.g., STRING (null):=null; rather than be writing constructs that appear to do something other than a null range, e.g., X(1..0). For ranges that the lower bound is greater than the upper bound, the programmer should get an error if the object is static and the code should raise an exception for the dynamic case. Erroneous constructs that appear to be correct code are just too easy to overlook in a large scale development. The attributes for null ranges need to be consistently defined--and not raise exceptions, e.g., FIRST=LAST=LENGTH=>0, regardless of the range expressed, 1..0 or -2..-3.

A problem exists with the example on 4.3.2 #14 with the comment "in particular, for N=0". The intent of the statement is not clear as to the number of cells being created. A reasonable user could intuit that either 0, 1, or 2 cells are created or that an error is generated.

IMPORTANCE: IMPORTANT

Program maintenance is 40% of costs of large scale weapons systems. The language should not allow constructs that appear to be wrong, i.e., the null range cases.

CURRENT WORKAROUNDS:

In programming standards, do not allow null strings, ranges, or components.

POSSIBLE SOLUTIONS:

Improve the method for expressing null ranges, e.g., "Null" attribute, rather than create error conditions to be raised at runtime.

INCONVENIENT HANDLING OF SCALAR TYPES THAT ARE CYCLIC IN NATURE

DATE: September 25, 1989

NAME: Stephen J. Wersan, Ph.D.

ADDRESS: Code 3561
NAVWPNCEN
China Lake, CA 93555

TELEPHONE: (619) 939-3120,
Autovon 437-3120
E-mail: WERSAN%356VAX.DECNET2NWC.NAVY.MIL

ANSI/MIL-STD-1815A REFERENCE: 3.5

PROBLEM:

Inconvenient handling of scalar types that are cyclic in nature.

IMPORTANCE: IMPORTANT

Cyclic types arise in many areas. Two important examples are types used to deal with time and compass direction (azimuth). Some of the messy trivia arising in the embedded world concerns handling such types represented by roll-over counters, and the like. This is probably closely related to unsigned integer types, and anything done in this area should mesh with what Ada 9X does there.

CURRENT WORKAROUNDS:

The use of the pred and succ attributes must be avoided. Special functions must be written to replace them.

POSSIBLE SOLUTIONS:

Extend the declaration of a scalar type to include the optional word "cyclic." Example:

> **type** DAY **is** cyclic (SUN, MON, TUE, ..., SAT);
>
> Having done so, the meanings of the attributes succ, pred and val would be altered so that
>
> DAY'succ(SAT) = SUN **and** DAY'pred(SUN) = SAT
>
> **while** DAY'val(9) = DAY'val(9 **mod** 7) = TUE.
>
> The attributes first, last and pos would continue to refer to the declaration ordering of the members of the type.
>
> In the embedded world, users of cyclic type will want to be informed when rollover or underflow occur. A predefined exception should be introduced for this purpose.

ENUMERATION LITERAL INTEGER CODES

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 3.5.1, 13.3

PROBLEM:

Once an Enumeration literal to integer code mapping has been established via an enumeration-representation-clause, there is no portable provision for retrieving the codes.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None that are guaranteed to be portable.

```
e.g. type colors is (Red, White, Blue);
        for colors use (3,7,9);
        function convert is new Unchecked_Conversion(Colors,Integer);
```

Note that "Convert" allows the code retrieval.

POSSIBLE SOLUTIONS:

Include an attribute of enumeration types, "'Code" as : Colors'Code(Red). This would evaluate to 3. If there is not representation clause, then the returned value would default to the position number, e.g., Color'Code(Red) would give 0.

ATTRIBUTES FIRST AND LAST FOR NULL RANGES ARE DEFINED RATHER ODDLY

DATE: June 12, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 3.5:3-4, 3.6.2

PROBLEM:

We use the 'Picture 9' example in the Dear Ada column in the ACM's Ada Letters publication -- May/June 1989 edition --to illustrate this similar but really different problem. There are internal inconsistencies in the definitions of 'first and 'last and 'length for null ranges.

Given:

```
function 1c ( S : string) return String is
  I : Positive := S'First ...
```

With the string declaration x : string(1..10); constraint error is raised on a call like 1c(x(-1)); the exception is raised on the attempt to assign the attribute S'first to a positive variable.

x(-1..-7) is a null string; its lower bound is greater than its upper bound. The number of elements in this string is zero.

We have no complaints so far. X(-1..-7) 'first is -1,
'last is -7 and
'length is 0.

What we have is a language in which 'last - 'first + 1 = 'length cannot be asserted to be true, not even limiting one's attention to those cases in which arithmetic exceptions are not raised. This is odd. This oddity is at least alluded to within the language definition -- inside a parenthetical statement within a note section, 3.6.2:14.

Another observation that strikes us as odd is that the attributes 'first and 'last of null ranges are not going to yield the same results in general. x(-1..-7)'last /= x(-2..-3)'last, even though both ranges are null.

The problem in the Ada Letters example arises because the virtual programmer assumed that 'first would always return a legal value of the type that is declared to be the index type of the array.

It should in fact be reasonable to assume that situations resembling

```
type t is array (positive range <>) of character;
x  : t(lb..ub);
```

I : positive := x'first

are not going to raise problems of the sort noted in the Ada Letters column. Yet our Ada language definition has problems here. The problem is not in the virtual programmer; the problem is in the language definition.

IMPORTANCE: IMPORTANT

Glitches like this contribute to the language's well-deserved idiosyncratic reputation.

CURRENT WORKAROUNDS:

In the Ada Letters example, change the type of I to integer; in the general case, reduce the amount of range checking done.

POSSIBLE SOLUTIONS:

The fix for the situation noted in the Ada Letters column is to fix the statement in 3.5:4 that allows checking for range compatibility to be postponed until attributes are used. But that doesn't address the issues dealing with the relationship of 'first, 'last, and 'length nor the unpredictable values for 'first and 'last of null ranges.

One should really consider changing the definition of 'first and 'last for null ranges. We can believe that this definition was made to allow compilers to be more easily implemented; if so, we appreciate the sentiment. What one might really want instead, though, is for 'first and 'last to return for null ranges a constant that serves the same purpose for discrete types as null does for access types. Much like the idea of NaN in the IEEE floating point standard.

An intermediate level solution might be for 'first and 'last to raise an exception when used with null ranges; we suspect that would be worse than the current situation, though.

IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (I)

DATE: September 21, 1989

NAME: Ivan B. Cvar (Canadian AWG #001)

ADDRESS: PRIOR Data Sciences Ltd.
240 Michael Cowpland Drive,
Kanata, Ontario, Canada,
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 3.5.4, 3.5.7, 10.3.9, 13.7.2, 3.8, 4.1.4, 13.1, 2.8, 13.9, 10.4, 10.1, 10.6, 3.2.1, 1.6

PROBLEM:

Implementation Options Lead to Non-Portability and Non-Reusability.

DISCUSSION:

The LRM allows many implementation options and this freedom has lead to numerous "dialects" of Ada. As programs are written to rely on the characteristics of a given implementation, non-portable Ada code results. Often, the programmer is not even aware that the code is non-portable, because implementation differences amy even exist for the predefined language features. Further, it is sometimes not impossible to compile an Ada program with two different implementations of the same vendor's compiler.

Another kind of non-portability is that of the programmer's skills, The user interfaces to Ada compilers have become so varied that programmers find it very difficult to move from one Ada implementation to another, Not only does the command line syntax vary, but so do program library structures, library shareability between users, compiler capabilities, capacity limits. etc.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Significant amounts of code rewriting, recompilation, and testing must be done to get a given Ada program to compile and to run successfully using another compiler, if at all possible, even on the same host-target configuration. It is very difficult to write a truly portable Ada program.

Another possible solution to porting an Ada program is for a customer to carefully choose a compiler to suit the given Ada program, or perhaps collaborate with a vendor to tailor the compiler to suit these needs.

Significant amounts of programmer retraining must occur when a different Ada compiler is used.

POSSIBLE SOLUTIONS:

Disallow or severely limit the number of allowed implementation options. The LRM should be changed to mandate more requirements that must be met by all implementations, and where freedoms are still allowed, restrict the implementation choices to a limited set of choices.

REQUIREMENTS:

	LRM Reference	Description
1.	LRM 3.5.4	The number of bits used to represent the predefined type INTEGER must be defined by the programmer of by the LRM, rather than being left to the implementation.
2.	LRM 3.5.4	Every implementation should be required to support both SHORT_INTEGER and LONG_INTEGER, with the bit sizes specified by the programmer or by the LRM.
3.	LRM 3.5.7	Every implementation should be required to support both SHORT_FLOAT and LONG_FLOAT, with the bit sizes specified by the programmer or by the LRM.
4.	LRM 10.3.9	Separate compilation of generic specifications and bodies must be made mandatory. Implementations should not be allowed to impose a restriction that the generic body be compiled along with the specification.
5.	LRM 13.7.2	The meaning of attribute 'ADDRESS must be clarified. For example, when applied to a program unit, the LRM must specify whether it refers to the program unit's entry point or the lowest (or highest) memory location occupied by the unit.
6.	LRM 13.7.2	The meaning of attribute 'SIZE must be clarified. For example, when applied to a type, the LRM must specify whether it refers to the minimum or to the actual number of bits used by the implementation to store a value of the type. When specified in a representation clause, the LRM should specify whether an implementation must use the stated "upper Bound" or whether it may ignore the representation clause and use a smaller size.
7.	LRM 3.8	The language must specify the relationship between access types and types SYSTEM.ADDRESS.
8.	LRM 4.1.4	The language must specify some common set of predefined attributes and not allow implementors to arbitrarily invent more new ones that can be used to circumvent the Ada rules.
9.	LRM 13.1	All representation clauses must be made mandatory, and fully defined.
10.	LRM 2.8	All predefined pragmas must be made mandatory, and fully defined.
11.	LRM 13.9	Pragma INTERFACE must be extended to allow the calling conventions (i.e. object format and other parameters) to be specified, or perhaps some other general mechanisms that would allow Ada to interface to other non-Ada components.
12.	LRM 10.4	The program library structure should be standardized.

13. LRM 10.4 — The packages provided, or allowed, in the runtime support library should be defined by the LRM to remove the wide variations between implementations.

14. LRM 10.1 — The command line syntax, or a set of standardized compiler switches should be mandated, to control debug status, code optimization, conditional compilation, or other requirements.

15. LRM 10.1 — The LRM should clarify the meaning of a main program and how it relates other library units as well as to the so-called environment task that invokes it, as well as to other tasks in the environment.

16. LRM 10.6 — The elimination of dead code, uncalled procedures, unused data, and the optimization performed must all be reported, and in a uniform manner.

17. LRM 3.2.1. — The detection and use of uninitialized variables must be standardized in the LRM to result in warning messages at compile time.

18. LRM 1.6 — Whenever an implementation chooses to ignore a portion of the program (eg. a pragma, representation clause, etc.), the implementation should be required to report a warning message.

19. LRM 1.6 — The LRM should define a standardized set of error messages.

20.

21.

22. (New) If it is impossible to remove all implementation options from the language, the ALRM should have a consolidated list of those optional areas. It should also be required that compiler vendors report the exact interpretation of these options. Appendix F lists the Implementation Dependant Characteristics but it is not complete. This appendix should be extended to include an entry for every remaining use of the phrases:

a) implementation dependent
b) machine dependent
c) system dependent
d) undefined
e) not defined
f) not required
g) may

IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (II)

DATE: October 1, 1989

NAME: Ivan B. Cvar (Canadian Ad Working Group)

ADDRESS: PRIOR Data Sciences Ltd.,
240 Michael Cowpland Drive,
Kanata, Ontario, Canada,
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 3.5.4, 3.5.7, 10.3.9, 13.7.2, 13.2, 13.8, 14.1.4, 13.1, 2.8, 11.7, 13.9, 10.4, 10.1, 10.6, 1.6, 3.2.1, 9.6

PROBLEM:

Compiler implementation options lead to non-portability and non-reusability of Ada code. The language allows many implementation options and this freedom has lead to numerous "dialects" of Ada. As programs are written to rely on the characteristics of a given implementation, non-portable Ada code results. Often, the programmer is not even aware that the code is non-portable, because implementation differences may even exist for the predefined language features. Further, it is sometimes not possible to compile an Ada program with two different implementations of the same vendor's compiler.

Another kind of non-portability is that of the programmer's skills. The user interfaces to Ada compilers have become so varied that programmers find it very difficult to move from one Ada implementation to another. Not only does the command line syntax vary, but so do program library structures, library shareability between users, compiler capabilities, capacity limits, etc.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

The use of non-portable predefined types, pragmas, attributes, representation clauses, etc. must be avoided, if possible.

Significant amounts of code rewriting, recompilation, and testing must be done to get a given Ada program to compile and to run successfully using another Ada compiler. In some circumstances it may not be possible, even on the same host-target configuration. It is very difficult to write a truly portable Ada program.

Another possible workaround to porting an Ada program is for a customer to carefully choose a compiler to suit a particular Ada program, or perhaps collaborate with a vendor to tailor the compiler to suit the current needs.

In some cases, significant amounts of programmer retraining must occur when using a different Ada compiler from another vendor.

POSSIBLE SOLUTIONS:

Disallow or severely limit the number of allowed implementation options. The LRM must be changed to mandate more requirements that must be met by all implementations, and where freedoms are still allowed, restrict the implementation choices to a limited set of choices.

Samples of such implementation options and their possible solutions follows:

ITEM	LRM REFERENCE	DESCRIPTION
1.	LRM 3.5.4	The number of bits used for type INTEGER must be either defined by the language, or must be definable by the programmer, rather than being left to the implementation. Perhaps a representation clause that allows the programmer to specify the exact 'SIZE or the 'MIN_SIZE in bits.
2.	LRM 3.5.4	The predefined types SHORT_INTEGER and LONG_INTEGER must be supported by all implementations. Also, the number of bits used for type SHORT-INTEGER and LONG_INTEGER must be either defined by the language,or must be definable by the programmer, rather than being left to the implementation. Perhaps a representation clause that allows the programmer to specify the exact 'SIZE or the 'MIN-SIZE in bits.
3.	LRM 3.5.7	The predefined types SHORT FLOAT and LONG_FLOAT must be supported by all implementations. Also, the number of bits used for type SHORT_FLOAT and LONG_FLOAT must be either defined by the language, or must be definable by the programmer, rather than being left to the implementation. Perhaps a representation clause that allows the programmer to specify the exact 'SIZE or the 'MIN SIZE in bits.
4.	LRM 10.3.9	Separate compilation of generic specifications, bodies,and subunits must be made mandatory for all implementations. Implementations must not be allowed to impose a restriction that the generic body or its subunits be compiled along with the generic specification.
5.	LRM 13.7.2	The meaning of attribute 'ADDRESS must be clarified. For example, when applied to a program unit, the LRM must specify whether it refers to the program unit's entry point or to the lowest (or highest) memory location occupied by the unit. In particular, when applied to a subprogram, the attribute must yield the address of the subprogram's entry point.
6.	LRM 13.7.2 and 13.2	The meaning of attribute 'SIZE must be clarified. For example, when applied to a type, and the LRM must specify whether it refers to some minimum or the actual number of bits used by the implementation to store a value of the type. Also, when specified in a length clause, the LRM must specify whether an implementation must use the stated "upper bound" as the actual size or whether it may ignore programmer's length clause and use a smaller size.

7. LRM 3.8 and 13.7

The language must specify the relationship between access types and type SYSTEM.ADDRESS. Thus, the effect of passing an access type or doing an Unchecked_Conversion between these two types would be guaranteed, particularly when interfacing to another language, like assembly language.

8. LRM 4.4.4

The language must specify some common set of predefined attributes, and not allow implementors to arbitrarily invent more new ones that can be used to circumvent the Ada rules.

9. LRM 13.1

All implementations must be required to support all representation clauses, and they must be fully defined.

10. LRM 2.8 and 11.7

All implementations must be required to support all predefined pragmas, and they must be fully defined.

For example, the effect of pragma SUPPRESS must be defined to specify whether it precludes the inclusion of ADDITIONAL object code to perform the required checks; or whether it prohibits the check for the error condition as well as prohibiting the raising of the exception; or whether it just prohibits the detection of the exception (perhaps at the hardware level).

In certain real time applications, exceptions can be safely ignored (due to the self-correcting nature of feedback loops). Processing constraints may make it undesirable to raise an exception, even if it is ignored. In this case, pragma SUPPRESS (or a similar pragma) must be defined to eliminate the possibility of ever raising an exception.

11. LRM 13.9

Pragma INTERFACE must be extended to allow the calling conventions (that is, object format and other parameters) to be specified, or perhaps some other general mechanism could be provided to allow an Ada program to interface to other non-Ada components written in a language unknown or not supported by the implementation.

12. LRM 10.4

The LRM mandates the presence of a program library, but does not define its structure or its interface. These must be standardized.

13. LRM 10.4

The packages that are provided, or allowed, in the runtime support library must be defined by the LRM to remove the wide variations between implementations.

14. LRM 10.1

The language must clarify the meaning of a main program and how it relates other library units as well as to the so-called environment task that invokes it, and to the other tasks in the environment.

15. LRM 10.6

The elimination of dead code, uncalled procedures, unused data, and the optimization performed must all be reported by an implementation, perhaps in response to a pragma.

16. LRM 1.6

Whenever an implementation chooses to ignore a portion of the program

(for example, a pragma, representation clause, etc.), the implementation must be required to report a warning message.

17. LRM 3.2.1 The detection and use of un-initialized variables must be standardized in the language to result in warning messages at compile time.

18. LRM 9.6 The language must specify whether a delay statement with a zero value either allows, or forces, scheduling to occur.

UNSIGNED INTEGER TYPES

DATE: October 1, 1989

NAME: Ivan B. Cvar, Canadian Ada Working Group

ADDRESS: PRIOR Data Sciences Ltd.,
240 Michael Cowpland Drive,
Kanata, Ontario, Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 3.5.4

PROBLEM:

There is a need for predefined unsigned integer types that use all of the bits of the type. The predefined integer types of size n bits cannot be used to represent values in the range 2**(n-1)..(2**n)-1 because they must be symmetrical about zero, thus requiring a sign bit. Also, attempts to assign a large positive value exceeding 2**(n-1)-1 to a variable of the type will raise an exception.

This need arises particularly when performing hardware address calculations or other numeric operations on non-negative data which occupies the full bit width of the type.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

In some implementations, an integer type with a larger number of bits may be used, if one is available. Otherwise, pragma INTERFACE to another language may be used.

Enumeration types may be used in some circumstances, but algorithms do not always lend themselves to the operators available for enumeration types.

POSSIBLE SOLUTIONS:

The language must define predefined unsigned integer types.

OPTIONAL INTEGER TYPES

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 3.5.4(7)

PROBLEM:

The LRM currently precludes *nonstandard integer types, e.g., unsigned.* This forces the compiler to translate integer subtypes into representations that are longer than otherwise necessary, and in some cases forces the use of otherwise unnecessary representation clauses. Several people have already submitted requests for the special case of unsigned, but the problem is more general.

IMPORTANCE: IMPORTANT

Otherwise will rely on representation clauses, introducing another potential source of errors.

CURRENT WORKAROUNDS:

- Representation clauses
- Types defined in vendor-supplied packages

Neither of these is portable.

POSSIBLE SOLUTIONS:

Replace 3.5.4(7) with the following:

The predefined integer types include the type INTEGER. An implementation may also have predefined types, such as SHORT_INTEGER and LONG_INTEGER, subject to the following restrictions.

No such type whose name begins with BYTE_ or with UNSIGNED may have negative values. The range

of any other such type whose name contains INTEGER must be symmetrical about zero, excepting an extra negative value that may exist in some implementations, e.g., two's complement.

Any such type whose name ends in _n, for a decimal number, must be exactly n bits long. The type LONG_INTEGER (SHORT_INTEGER), if it exists, must not be shorter (longer) than the type INTEGER.

The vendor is free to include or omit any of the above independently, except for the type INTEGER, which is mandatory.

UNSIGNED ARITHMETIC

DATE: September 25, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3.5.4(7)

PROBLEM:

The RM does not provide for implementation of unsigned integer types. Although many machines have unsigned arithmetic instructions available, the semantics of Ada currently prevent the efficient calculation of checksums, hash codes, pseudo-random numbers, and full range address arithmetic using those instructions.

The need for unsigned integers in Ada is clear; many users have applications (which are trivial to implement *in many other languages*) that cannot presently be written directly in Ada. In response to their customers' expressed desires, a number of vendors have provided some form of support for unsigned integers which conforms to the current standard, but they are only partially successful in providing the desired functionality and their approaches are quite diverse. As a consequence, a uniform approach which addresses the major uses of unsigned numbers should be incorporated into Ada9x.

The goals for unsigned integer types include:

1 providing an extended maximum non-negative integer range which fully exploits the available hardware (and which allows full range address arithmetic when appropriate),

2. providing straightforward and efficiently-implementable logical operations (including shifts, rotates, and masks) on all bits of unsigned types,

3. providing numeric literals in arbitrary bases (so that representations appropriate to a given architecture may be chosen for bit-level operations), and

4. providing efficient support for modular arithmetic of arbitrary range (which allows checksums, hash functions, and pseudo-random number generators which generate all possible bit patterns in closed cycles to be cleanly written in Ada).

These requirements have been derived from extensive discussions with end users of the language and would satisfy most, if not all of their expectations. There were two major subsets of properties desired by different groups of users:

1. modular arithmetic required and logical operations seen as essential, and

2. range-checked arithmetic expected, logical operations relatively unimportant.

The first group were very vocal and definite about their needs, and the second (and much smaller) group were generally less so. The current proposal, which omits range-checked arithmetic, would, I believe, be viewed as acceptable by the majority of those who expressed an opinion.

A number of people have suggested that logical operations can be provided using packed Boolean arrays and instances of Unchecked_Conversion in order to convert back and forth from unsigned types to the array type. There are a number of reasons why this approach was rejected here. Firstly, the intent of this proposal is to encourage implementers to use the obvious machine instructions for efficiency. If conversion to Boolean arrays are used, it is very clear that some implementations will not use the desired instructions, because that part of the language has already been implemented in Ada83. Secondly, the clarity of expression is better when appropriate based numeric literals are used. Last, but not least, I tried the use of Boolean arrays and unchecked conversion and gave it up as too clumsy and inefficient of programmers' time, besides being error-prone because the notation is not well-suited to the semantics of the intended usage.

IMPORTANCE: ESSENTIAL

Current workarounds are generally unsatisfactory and highly non-portable. The Ada language is clearly deficient in this area with regard to its support for its intended application domain: embedded, real-time systems.

CURRENT WORKAROUNDS:

DEC defines its unsigned types in package SYSTEM. There are two "genuine" unsigned integer types called UNSIGNED_BYTE and UNSIGNED_WORD plus a signed integer type called UNSIGNED_LONGWORD and a record type called UNSIGNED_QUADWORD which has two components of type UNSIGNED_LONGWORD. Static subtypes of the (signed) type UNSIGNED_LONGWORD are provided. No discussion of further derivability of these types is provided. By the absence of explicit statements, it may be inferred that the arithmetic on these unsigned types is not modular.

For each unsigned integer type, the DEC implementation provides logical operators and a corresponding constrained subtype of a common unconstrained packed BOOLEAN array type and conversions to and from that subtype and the unsigned types. For example:

type BIT_ARRAY **is array** (INTEGER **range** <>) **of** BOOLEAN; **pragma** PACK
(BIT_ARRAY); -- There must be exactly 1 bit for each BOOLEAN -- component, which DEC kindly provides for -- packed unconstrained arrays of BOOLEANs.

Question: What is the weight of bit i? Is it 2**i or 2**(nn - 1 - i)?
Answer: It is implementation-dependent.

BIT_ARRAY should probably not be a standard part of the definition of unsigned types, because the added functionality (indexing and catenation) is of marginal value, the order of bits is implementation-dependent, and users can implement it themselves if they so desire (using UNCHECKED_CONVERSION). Implementers may decide to provide it anyway, of course. The functional capabilities of BIT_ARRAY, other than catenation, are available for the unsigned types proposed above, and the literals are much nicer. For instance (taking a "little-endian" view):

```
if (Logical_Shift(Modular_Value, -4) and 1) = 1 then ... -- bit 4 is on
```

or, perhaps more efficiently and more clearly:

```
Bit_4 : constant := 2#10000#; if (Modular_Value and Bit_4) = Bit_4 then ... -- bit 4 is on.
```

Alsys defines its unsigned types in package UNSIGNED (available only for the PC compiler). It defines two unsigned integer types, type BYTE (range 0 .. 255) and type WORD (range 0 .. 65535).

Their representations in memory are context dependent. As record and array components, they occupy 8 and 16 bits, respectively; as scalar objects, they both occupy 16 bits.

Their arithmetic is modular and their subtypes have range-checking, but they are not "properly" derivable, that is to say, types derived from them will be ordinary integer types, because it is asserted that their base types are predefined integer types, rather than the types themselves.

Instantiations of UNCHECKED_CONVERSION are provided to and from other integer types, although one would expect explicit conversions to be sufficient. Unchecked conversion from CHARACTER to BYTE is provided, but (curiously) no function is provided to convert BYTE to CHARACTER (which ought to raise CONSTRAINT_ERROR if the high bit is on). A user could (apparently) instantiate UNCHECKED_CONVERSION on these types himself to remedy the oversight. Functions LSB, MSB, and MAKE_WORD which extract the least or most significant BYTE of a WORD and construct a WORD from two BYTES are provided; they could have been written in Ada.

Verdix provides a package called Unsigned_Types defining true 8-, 16-, and 32-bit unsigned integers without logical operations. (They say to use it at your own risk, since it is not valid Ada). They also provide a separate package called Iface_Bits for bit-wise logical operations on type Integer. Contrary to what might be expected, they are named Bit_And, Bit_Or, Bit_Xor, and Bit_Neg, rather than the usual operator symbols.

POSSIBLE SOLUTIONS:

The principal language issue is whether unsigned types can be provided at all and if so, how. Currently, there are two relevant AI's: AI-00402 and AI-00597. AI-00402, which has been unanimously approved by the ARG, but has not yet been seen by WG9 or AJPO, simply affirms (as a ramification) that unsigned integer types are not predefined. AI-00597 is a work item that will directly address how unsigned integers can be provided, if at all. The ARG has decided to take no action on this AI until the URG has determined that AI-00597 is indeed proposing a necessary and useful interpretation of the standard, i.e., one that is required to support a "necessary and useful" implementation
of unsigned integers.

Technically, the main problem is LRM 3.5.4(7), which says in part:

> The range of each of these [predefined integer] types must be symmetric about zero, excepting an extra negative value which may exist in some implementations.

Robert Dewar and Paul Hilfinger have suggested that this sentence might be interpreted to allow the introduction of implementation-defined integer types in package SYSTEM or elsewhere. Such types would not be predefined integer types and thus would not be used by the implementation in the derivations of 3.5.4(4-6), although other types might be derived from them and subtypes might be defined based on them.

Bryce Bardin has defined a standard package for unsigned integer types which takes advantage of this interpretation. It takes the view that unsigned integer types are integer types in every other respect than not participating in implicit derivations. In particular, they have as a subset of their predefined operations the same operations provided for the predefined integer types, although the meaning of those operations are different. A primary of a non-static universal expression can be implicitly converted to an unsigned integer type. In addition, they match generic formal discrete and integer types. (E.g., TEXT_IO.INTEGER_IO can be instantiated for these types.) Finally, it is proposed that the declaration of subtypes of unsigned types and types derived from unsigned types will behave the same as for predefined and user-defined integer types. In particular, it is intended that LRM 3.5.5(12) apply after modification by replacing "predefined" by "implementation-defined" at its second occurrence: "..., the predefined operators of an integer type deliver results whose range is defined by the parent [implementation-defined] type; such a result need not belong to the declared subtype, in which case an attempt to assign the result to a variable of the integer subtype raises the exception CONSTRAINT_ERROR."

The additional operations on these types include bit-wise logical operators. When reading the description of ARITHMETIC_SHIFT, remember that the unsigned types are non-negative (i.e., have no sign bit), which leads to zero filling of the most significant bit. In accordance with the recommendation of AI-00387, the operations on unsigned types always raise CONSTRAINT_ERROR, rather than NUMERIC_ERROR when they cannot deliver the correct result.

In what follows, asides and comments are enclosed in square brackets ([]).

START OF PROPOSAL

DRAFT PROPOSAL ON UNSIGNED INTEGER TYPES

An implementation-defined unsigned integer type definition defines an integer type whose set of values include exactly the specified range, where the lower bound is zero and the upper bound is either 2**n - 1 or 2**n - 2 for some positive integer n. The base type of such a type is the type itself.

Operations on implementation-defined unsigned integer types include all of the operations on integer types plus the predefined logical operators and the highest precedence operator "not". The arithmetic operators have their conventional meaning, but the arithmetic is modular rather than range-checked as it is for the predefined integer types. The logical operators have their conventional meaning as applied to unsigned integers viewed as arrays of 1-bit numeric values which represent boolean values (with 0 corresponding to FALSE and 1 corresponding to TRUE). (Note: based integer literals are available for defining values in conventional formats, e.g., hexadecimal.) Additional operations for bit-wise arithmetic and logical shifts and rotations are defined in the package specification given below.

For every integer type or subtype T, the following (implementation-defined) attribute is defined:

T'MODULAR — Yields the value TRUE if T is an unsigned integer type with modular arithmetic; yields the value FALSE otherwise. The value of this attribute is of the predefined type BOOLEAN.

[This attribute facilitates the usage of modular types in generic units.]

Every implementation of should provide at least one unsigned integer type with modular arithmetic.

The implementation-defined unsigned types using modular arithmetic should include the type MODULAR_nn, where nn represents an integer value equal to both MODULAR_nn'SIZE and

INTEGER'SIZE. An implementation may also have other implementation-defined unsigned integer types using modular arithmetic with names of the same form which have different sizes. The arithmetic operations on these types are performed modulo 2**nn (for 2's complement machines) or modulo 2**nn - 1 (for 1's complement machines).

[It would be symmetric to have a predefined integer type named INTEGER_nn instead of INTEGER here. It would be less universal than INTEGER, but directly comparable to MODULAR_nn, and directly tied to a specific hardware representation by a standard name.]

[The explicit declaration of unsigned types cannot be given in Ada because their base types are not implicitly derived from predefined integer types as required by 3.5.4(4-6). Because of that, their declarations are given here in the style of package STANDARD (see Annex C).]

The following outlines the specification of the package MODULAR_NUMBERS, containing all implementation-defined unsigned integer type declarations. The corresponding package body is implementation-defined and is not shown.

The operators that are predefined for the types declared in the package MODULAR_NUMBERS are given in comments since they are implicitly declared. Italics are used [will be used] for pseudo-names of anonymous types and for undefined information.

```
package MODULAR_NUMBERS is

    type MODULAR_nn is implementation-defined;

    -- Note:

    --   MODULAR_nn'FIRST = 0
    --   MODULAR_nn'LAST = 2**nn - 1 or 2**nn - 2
    --   MODULAR_nn'BASE'FIRST = MODULAR_nn'FIRST = 0
    --   MODULAR_nn'BASE'LAST = MODULAR_nn'LAST
    --   MODULAR_nn'PRED(MODULAR_nn'FIRST) = MODULAR_nn'LAST
    --   MODULAR_nn'SUCC(MODULAR_nn'LAST) = MODULAR_nn'FIRST

    for MODULAR_nn'SIZE use nn;

    -- The predefined operators for this type are as follows:

    -- function "="      LEFT, RIGHT : MODULAR_nn) return BOOLEAN; -- function "/="
                         (LEFT, RIGHT : MODULAR_nn) return BOOLEAN;
    -- function "<"      (LEFT, RIGHT : MODULAR_nn) return BOOLEAN; -- function "<="
                         (LEFT, RIGHT : MODULAR_nn) return BOOLEAN;
    -- function ">"      (LEFT, RIGHT : MODULAR_nn) return BOOLEAN; -- function ">="
                         (LEFT, RIGHT : MODULAR_nn) return BOOLEAN;
    -- function "+"      (RIGHT : MODULAR_nn) return MODULAR_nn;
    -- function "-"      (RIGHT : MODULAR_nn) return MODULAR_nn;
    -- function "abs"    (RIGHT : MODULAR_nn) return MODULAR_nn;

    -- function "+"      (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
    -- function "-"      (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
    -- function "*"      (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
```

```
-- function "/"         (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
-- function "rem"       (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
-- function "mod"       (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;

-- function "**"        (LEFT  : MODULAR_nn;
--                       RIGHT : INTEGER) return MODULAR_nn;

function "and"          (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
function "or"           (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;
function "xor"          (LEFT, RIGHT : MODULAR_nn) return MODULAR_nn;

function "not"          (LEFT        : MODULAR_nn) return MODULAR_nn;

function ARITHMETIC_SHIFT (ITEM : MODULAR_nn; BITS: INTEGER) return
                           MODULAR_nn;

--      If BITS >= 0 or BITS < 0, returns ITEM left or right arithmetically shifted (with zero
        fill) by abs(BITS) bits, respectively. Raises CONSTRAINT_ERROR if a bit would be
        shifted off the left end.

function LOGICAL_SHIFT(ITEM : MODULAR_nn; BITS : INTEGER) return MODULAR_nn;

--      If BITS >= 0 or BITS < 0, returns ITEM left or right logically shifted (end off) by
        abs(BITS) bits, respectively.

function ROTATE (ITEM : MODULAR_nn; BITS : INTEGER) return MODULAR_nn;

--      If BITS >= 0 or BITS < 0, returns ITEM left or right rotated (end around) by abs(BITS)
        bits, respectively.
```

[The following two functions, REM_OF_SUM and REM_OF_PRODUCT, are needed to allow the construction of types derived from MODULAR_nn, but with smaller ranges, having the desired semantics. One possible package for this purpose is discussed below.]

```
function REM_OF_SUM (ADDEND,
                                   AUGEND,
                                   DIVISOR : MODULAR_nn) return MODULAR_nn;
-- Returns MODULAR_nn((Anonymous(ADDEND) + Anonymous(AUGEND)) rem
Anonymous(DIVISOR)), where Anonymous is some integer type for which 2 *
MODULAR_nn'LAST is within the range of values.

function REM_OF_PRODUCT (MULTIPLIER,
                                   MULTIPLICAND,
                                   DIVISOR : MODULAR_nn) return MODULAR_nn;

-- Returns MODULAR_nn((Anonymous(MULITPLIER) * Anonymous(MULTIPLICAND)) rem
Anonymous(DIVISOR)), where Anonymous is some integer type for which MODULAR_nn'LAST
** 2 is within the range of values.

-- The following type facilitates the declaration of unsigned types of maximum range:
```

```
type LARGEST_MODULAR_TYPE is
new implementation_defined_modular_type;
```

[These could also be declared using subtypes and renaming of operations, but the derived type approach used here is simpler and therefor clearer to the user.]

```
end MODULAR_NUMBERS;
```

[In analogy with:

```
type T is range SYSTEM.MIN_INT .. SYSTEM.MAX_INT;
```

for symmetric integer types, unsigned integer types of maximum range can be declared (in portable syntax) by:

```
type T is new LARGEST_MODULAR_TYPE;]
```

END OF PROPOSAL

The following material amplifies the semantics of the proposal.

Operational Semantics for Unsigned Types

The following package demonstrates the semantics desired for the implementation-dependent unsigned types defined in the draft proposal.

-- Modular_Numbers demonstrates (most of) the desired properties of implementation-defined unsigned numbers having modular arithmetic.

-- In particular, it shows the behavior required for the arithmetic, relational, and logical operations on type Modular. In addition, it provides analogs for the 'Pred and 'Succ attributes.

-- It was considered unnecessary to simulate the other attributes and the Shift and Rotate operations since they are expected to reflect the usual properties of the underlying hardware operations.

-- N. B.: This simulation is also not realistic with regard to the representation of the type Modular.

```
generic

    type Basis is range <>;          -- Any predefined or  user-defined integer type or subtype

    Modulus : in Basis;              -- Any positive value less than Basis'Last/2

    type Extended is range <>;       -- Any integer type with a range such that Extended'Last
                                     >= (Modulus - 1)**2
```

```
        type Logical is range <>;       -- Any integer type with a range at least 0 .. Modulus -
                                        1 for which bit-level logical operations are available.
        with function "and" (Left, Right : Logical) return Logical is <>;
        with function "or" (Left, Right : Logical) return Logical is <>;
        with function "xor" (Left, Right : Logical) return Logical is <>;
        with function "not" (Right : Logical) return Logical is <>;

package Modular_Numbers is

        type Modular is new Basis range 0 .. Modulus - 1;

-- Note:

-- Modular'First = 0

-- Modular'Last = Modulus - 1

-- Modular'Base'First = Modular'First = 0
-- (Not true in this simulation)

-- Modular'Base'Last = Modular'Last
-- (Not true in this simulation)

-- Modular'Pred(Modular'First) = Modular'Last
-- (Not true in this simulation)

-- Modular'Succ(Modular'Last) = Modular'First
-- (Not true in this simulation)

-- The predefined operators for this type are as follows:

-- function "="         (Left, Right : Modular) return Boolean;
-- function "/="        (Left, Right : Modular) return Boolean;
-- function "<"         (Left, Right : Modular) return Boolean;
-- function "<="        (Left, Right : Modular) return Boolean;
-- function ">"         (Left, Right : Modular) return Boolean;
-- function ">="        (Left, Right : Modular) return Boolean;

-- function "+"         (Right : Modular) return Modular;
-- function "-"         (Right : Modular) return Modular;
-- function "abs"       (Right : Modular) return Modular;

-- function "+"         (Left, Right : Modular) return Modular;
-- function "-"         (Left, Right : Modular) return Modular;
-- function "*"         (Left, Right : Modular) return Modular;
-- function "/"         (Left, Right : Modular) return Modular;
-- function "rem"       (Left, Right : Modular) return Modular;
-- function "mod"       (Left, Right : Modular) return Modular;

-- function "**"        (Left  : Modular; Right : Integer) return Modular;
```

```
        -- function "and"          (Left, Right : Modular) return Modular;
        -- function "or"           (Left, Right : Modular) return Modular;
        -- function "xor"          (Left, Right : Modular) return Modular;

        -- function "not"          (Right : Modular) return Modular;

        -- The following functions are cyclic substitutes for the  attributes Modular'Pred and Modular'Succ,
        respectively.

        -- function Pred           (Right : Modular) return Modular;
        -- function Succ           (Right : Modular) return Modular;

        -- The following functions are not implemented:

        -- function Arithmetic_Shift (Item : Modular_nn;
        -- Bits Integer) return Modular_nn;
        -- function Logical_Shift (Item : Modular;
        -- Bits : Integer) return Modular_nn;
        -- function Rotate (Item : Modular_nn;
        -- Bits : Integer) return Modular_nn;
        -- function Rem_of_Sum (Addend, Augend, Divisor : Modular) return Modular;
        -- function Rem_of_Product (Multiplier, Multiplicand, Divisor : Modular) return Modular;

end Modular_Numbers;


package body Modular_Numbers is

        function "-"            (Right : Modular) return Modular is
        begin
                if Modulus = 1 then
                return 0;
        else
                return (Modular'Last - Right) + 1;
        end if;
end "-";

function "+"    (Left, Right : Modular) return Modular is
begin
        return Modular((Extended(Left) + Extended(Right))
                                        mod Extended(Modulus));
end "+";

function "-"    (Left, Right : Modular) return Modular is
begin
        if Right > Left then
                return ((Modular'Last - Right) + Left) + 1;
        else
                return Modular(Basis(Left) - Basis(Right));
        end if;
end "-";
```

```
function "**"     (Left, Right : Modular) return Modular is
        Product : Modular;
begin
        return Modular((Extended(Left) * Extended(Right))
                                        mod Extended(Modulus));
end "**";

function "***"  (Left  : Modular;
                 Right : Integer) return Modular is
        T : Modular;
begin
        if Modulus = 1 then
                return 0;
        else
                T := 1;
                for N in 1 .. Right loop
                                T := T * Left;
                end loop;
                return T;
        end if;
end "***";

function "and" (Left, Right : Modular) return Modular is
begin
        return Modular(Logical(Left) and Logical(Right));
end "and";

function "or"  (Left, Right : Modular) return Modular is
begin
        return Modular(Logical(Left) or Logical(Right));
end "or";

function "xor" (Left, Right : Modular) return Modular is
begin
        return Modular(Logical(Left) xor Logical(Right));
end "xor";

package Make is
        function Mask return Logical;
end Make;

package body Make is

        Mask_Constant : Logical;

        function Mask return Logical is
        begin
        return Mask_Constant;
        end Mask;
```

```
begin

        Mask_Constant := 0;
        for C in Modular loop
        Mask_Constant := Mask_Constant or Logical(C);
        end loop;

end Make;

function "not" (Right : Modular) return Modular is
begin
        return Modular(not Logical(Right) and Make.Mask);
end "not";

function Pred (Right : Modular) return Modular is
begin
        if Right = 0 then
                return Modular(Modulus - 1);
        else
                return Modular(Basis'Pred(Basis(Right)));
        end if;
end Pred;

function Succ (Right : Modular) return Modular is
begin
        if Right = Modular(Modulus - 1) then
                return 0;
        else
                return Modular(Basis'Succ(Basis(Right)));
        end if;
end Succ;

function Rem_of_Sum (Addend, Augend, Divisor : Modular)
        return Modular is
begin
        return Modular(
                (Extended(Addend) + Extended(Augend)) remExtended(Divisor));
end Rem_of_Sum;

function Rem_of_Product (Multiplier,
                                                Multiplicand,
                                                Divisor : Modular) return Modular is
begin
        return Modular(
                (Extended(Multiplier) * Extended(Multiplicand)) rem Extended(Divisor));
end Rem_of_Product;

begin
        if Extended'Last < Extended(Basis'Last) ** 2 then
                raise Program_Error;
        end if;
```

```
end Modular_Numbers;
```

Unsigned Types of Arbitrary Range

The following package can be used to provide modular unsigned types of arbitrary range by derivation from the unsigned types in package MODULAR_NUMBERS. It is directly implementable by any user.

```
generic

        type Basis is range <>;
        Modulus : in Basis;

        with function Rem_of_Sum (Addend, Augend, Divisor : Basis)
        return Basis is <>;
        with function Rem_of_Product (Multiplier,
                                                        Multiplicand,
                                                        Divisor : Basis) return Basis is <>;

        with function "and" (Left, Right : Basis) return Basis is <>;
        with function "or"  (Left, Right : Basis) return Basis is <>;
        with function "xor" (Left, Right : Basis) return Basis is <>;
        with function "not" (Right : Basis) return Basis is <>;

        with function Pred (Item : Basis) return Basis is Basis'Pred;
        with function Succ (Item : Basis) return Basis is Basis'Succ;

package Derived_Modular is

        type Modular is new Basis range 0 .. Modulus - 1;

        -- function "<"   (Left, Right : Modular) return Boolean;
        -- function "<="  (Left, Right : Modular) return Boolean;
        -- function ">"   (Left, Right : Modular) return Boolean;
        -- function ">="  (Left, Right : Modular) return Boolean;

        -- function "+"   (Right : Modular) return Modular;
         function "-"   (Right : Modular) return Modular;
        -- function "abs" (Right : Modular) return Modular;

        function "+"   (Left, Right : Modular) return Modular;
        function "-"   (Left, Right : Modular) return Modular;
        function "*"   (Left, Right : Modular) return Modular;
        -- function "/"   (Left, Right : Modular) return Modular;
        -- function "rem" (Left, Right : Modular) return Modular;
        -- function "mod" (Left, Right : Modular) return Modular;

        function "**"  (Left : Modular; Right : Integer)
                return Modular;
```

```
        function "and" (Left, Right : Modular) return Modular;
        function "or"  (Left, Right : Modular) return Modular;
        function "xor" (Left, Right : Modular) return Modular;
        function "not" (Right : Modular) return Modular;

        function Pred (Item : Modular) return Modular;
        function Succ (Item : Modular) return Modular;

end Derived_Modular;


package body Derived_Modular is

        function "-" (Right : Modular) return Modular is
        begin
                if Modulus = 1 then
                return 0;
        else
                return (Modular'Last - Right) + 1;
end if;
end "-";

        function "+" (Left, Right : Modular) return Modular is
        begin
                return Modular(Rem_of_Sum(Basis(Left), Basis(Right), Modulus));
        end "+";

        function "-" (Left, Right : Modular) return Modular is
        begin
                if Right > Left then
                                return ((Modular'Last - Right) + Left) + 1;
                else
                                return Modular(Basis(Left) - Basis(Right));
                end if;
        end "-";

        function "*" (Left, Right : Modular) return Modular is
        begin
                return Modular(Rem_of_Product(Basis(Left), Basis(Right), Modulus));
        end "*";

        function "**" (Left : Modular; Right : Integer) return Modular is
                Temp : Modular;
        begin
                if Modulus = 1 then
                                return 0;
                else
                                Temp := 1;
                                for N in 1 .. Right loop
                                        Temp := Temp * Left;
                                end loop;
```

```
                        return Temp;
                end if;
        end "***";

        function "and" (Left, Right : Modular) return Modular is
        begin
                return Modular(Basis(Left) and Basis(Right));
        end "and";

        function "or" (Left, Right : Modular) return Modular is
        begin
                return Modular(Basis(Left) or Basis(Right));
        end "or";

        function "xor" (Left, Right : Modular) return Modular is
        begin
                return Modular(Basis(Left) xor Basis(Right));
        end "xor";

        package Make is
                function Mask return Basis;
        end Make;

package body Make is

        Mask_Constant : Basis;

        function Mask return Basis is
        begin
        return Mask_Constant;
        end Mask;

begin

        Mask_Constant := 0;
        for C in Modular loop
                Mask_Constant := Mask_Constant or Basis(C);
        end loop;

end Make;

        function "not" (Right : Modular) return Modular is
        begin
                return Modular(not Basis(Right) and Make.Mask);
        end "not";


        function Pred (Item : Modular) return Modular is
        begin
                if Item = 0 then
                        return Modular'Last;
```

```
        else
                        return Modular(Pred(Basis(Item)));
        end if;
end Pred;

function Succ (Item : Modular) return Modular is
begin
        if Item = Modular'Last then
                        return 0;
        else
                        return Modular(Succ(Basis(Item)));
        end if;
end Succ;

end Derived_Modular;
```

Examples of the Behavior of Unsigned Types

The following are examples of the desired behavior of modular unsigned types (the base 2 values assume 2's complement representation):

The modulus is 3

```
+ 0 = 0 (+ 2#0# = 2#0#)
+ 1 = 1 (+ 2#1# = 2#1#)
+ 2 = 2 (+ 2#10# = 2#10#)

- 0 = 0 (- 2#0# = 2#0#)
- 1 = 2 (- 2#1# = 2#10#)
- 2 = 1 (- 2#10# = 2#1#)

0 + 0 = 0 (2#0# + 2#0# = 2#0#)
. . .
2 + 2 = 1 (2#10# + 2#10# = 2#1#)

0 - 0 = 0 (2#0# - 2#0# = 2#0#)
. . .
2 - 2 = 0 (2#10# - 2#10# = 2#0#)

0 * 0 = 0 (2#0# * 2#0# = 2#0#)
. . .
2 * 2 = 1 (2#10# * 2#10# = 2#1#)

0 / 0 = Constraint_Error
0 / 1 = 0 (2#0# / 2#1# = 2#0#)
0 / 2 = 0 (2#0# / 2#10# = 2#0#)
1 / 0 = Constraint_Error
1 / 1 = 1 (2#1# / 2#1# = 2#1#)
```

```
1 / 2 = 0 (2#1# / 2#10# = 2#0#)
2 / 0 = Constraint_Error
2 / 1 = 2 (2#10# / 2#1# = 2#10#)
2 / 2 = 1 (2#10# / 2#10# = 2#1#)

0 rem 0 = Constraint_Error
0 rem 1 = 0 (2#0# rem 2#1# = 2#0#)
0 rem 2 = 0 (2#0# rem 2#10# = 2#0#)
1 rem 0 = Constraint_Error
1 rem 1 = 0 (2#1# rem 2#1# = 2#0#)
1 rem 2 = 1 (2#1# rem 2#10# = 2#1#)
2 rem 0 = Constraint_Error
2 rem 1 = 0 (2#10# rem 2#1# = 2#0#)
2 rem 2 = 0 (2#10# rem 2#10# = 2#0#)

0 mod 0 = Constraint_Error
0 mod 1 = 0 (2#0# mod 2#1# = 2#0#)
0 mod 2 = 0 (2#0# mod 2#10# = 2#0#)
1 mod 0 = Constraint_Error
1 mod 1 = 0 (2#1# mod 2#1# = 2#0#)
1 mod 2 = 1 (2#1# mod 2#10# = 2#1#)
2 mod 0 = Constraint_Error
2 mod 1 = 0 (2#10# mod 2#1# = 2#0#)
2 mod 2 = 0 (2#10# mod 2#10# = 2#0#)

0 ** 0 = 1 (2#0# ** 2#0# = 2#1#)
. . .
2 ** 2 = 1 (2#10# ** 2#10# = 2#1#)

0 and 0 = 0 (2#0# and 2#0# = 2#0#)
. . .
2 and 2 = 2 (2#10# and 2#10# = 2#10#)

0 or 0 = 0 (2#0# or 2#0# = 2#0#)
. . .
1 or 1 = 1 (2#1# or 2#1# = 2#1#)
1 or 2 = Constraint_Error
2 or 0 = 2 (2#10# or 2#0# = 2#10#)
2 or 1 = Constraint_Error
2 or 2 = 2 (2#10# or 2#10# = 2#10#)

0 xor 0 = 0 (2#0# xor 2#0# = 2#0#)
. . .
1 xor 1 = 0 (2#1# xor 2#1# = 2#0#)
1 xor 2 = Constraint_Error
2 xor 0 = 2 (2#10# xor 2#0# = 2#10#)
2 xor 1 = Constraint_Error
2 xor 2 = 0 (2#10# xor 2#10# = 2#0#)

not 0 = Constraint_Error
not 1 = 2 (not 2#1# = 2#10#)
```

not 2 = 1 (not 2#10# = 2#1#)

Pred(0) = 2 (Pred(2#0#) = 2#10#)
Pred(1) = 0 (Pred(2#1#) = 2#0#)
Pred(2) = 1 (Pred(2#10#) = 2#1#)

Succ(0) = 1 (Succ(2#0#) = 2#1#)
Succ(1) = 2 (Succ(2#1#) = 2#10#)
Succ(2) = 0 (Succ(2#10#) = 2#0#)

The modulus is 4

+ 0 = 0 (+ 2#0# = 2#0#)
+ 1 = 1 (+ 2#1# = 2#1#)
+ 2 = 2 (+ 2#10# = 2#10#)
+ 3 = 3 (+ 2#11# = 2#11#)

- 0 = 0 (- 2#0# = 2#0#)
- 1 = 3 (- 2#1# = 2#11#)
- 2 = 2 (- 2#10# = 2#10#)
- 3 = 1 (- 2#11# = 2#1#)

0 + 0 = 0 (2#0# + 2#0# = 2#0#)
. . .
3 + 3 = 2 (2#11# + 2#11# = 2#10#)

0 - 0 = 0 (2#0# - 2#0# = 2#0#)
. . .
3 - 3 = 0 (2#11# - 2#11# = 2#0#)

0 * 0 = 0 (2#0# * 2#0# = 2#0#)
. . .
3 * 3 = 1 (2#11# * 2#11# = 2#1#)

0 / 0 = Constraint_Error
0 / 1 = 0 (2#0# / 2#1# = 2#0#)
0 / 2 = 0 (2#0# / 2#10# = 2#0#)
0 / 3 = 0 (2#0# / 2#11# = 2#0#)
1 / 0 = Constraint_Error
1 / 1 = 1 (2#1# / 2#1# = 2#1#)
1 / 2 = 0 (2#1# / 2#10# = 2#0#)
1 / 3 = 0 (2#1# / 2#11# = 2#0#)
2 / 0 = Constraint_Error
2 / 1 = 2 (2#10# / 2#1# = 2#10#)
2 / 2 = 1 (2#10# / 2#10# = 2#1#)
2 / 3 = 0 (2#10# / 2#11# = 2#0#)
3 / 0 = Constraint_Error
3 / 1 = 3 (2#11# / 2#1# = 2#11#)
3 / 2 = 1 (2#11# / 2#10# = 2#1#)

```
3 / 3 = 1 (2#11# / 2#11# = 2#1#)

0 rem 0 = Constraint_Error
0 rem 1 = 0 (2#0# rem 2#1# = 2#0#)
0 rem 2 = 0 (2#0# rem 2#10# = 2#0#)
0 rem 3 = 0 (2#0# rem 2#11# = 2#0#)
1 rem 0 = Constraint_Error
1 rem 1 = 0 (2#1# rem 2#1# = 2#0#)
1 rem 2 = 1 (2#1# rem 2#10# = 2#1#)
1 rem 3 = 1 (2#1# rem 2#11# = 2#1#)
2 rem 0 = Constraint_Error
2 rem 1 = 0 (2#10# rem 2#1# = 2#0#)
2 rem 2 = 0 (2#10# rem 2#10# = 2#0#)
2 rem 3 = 2 (2#10# rem 2#11# = 2#10#)
3 rem 0 = Constraint_Error
3 rem 1 = 0 (2#11# rem 2#1# = 2#0#)
3 rem 2 = 1 (2#11# rem 2#10# = 2#1#)
3 rem 3 = 0 (2#11# rem 2#11# = 2#0#)

0 mod 0 = Constraint_Error
0 mod 1 = 0 (2#0# mod 2#1# = 2#0#)
0 mod 2 = 0 (2#0# mod 2#10# = 2#0#)
0 mod 3 = 0 (2#0# mod 2#11# = 2#0#)
1 mod 0 = Constraint_Error
1 mod 1 = 0 (2#1# mod 2#1# = 2#0#)
1 mod 2 = 1 (2#1# mod 2#10# = 2#1#)
1 mod 3 = 1 (2#1# mod 2#11# = 2#1#)
2 mod 0 = Constraint_Error
2 mod 1 = 0 (2#10# mod 2#1# = 2#0#)
2 mod 2 = 0 (2#10# mod 2#10# = 2#0#)
2 mod 3 = 2 (2#10# mod 2#11# = 2#10#)
3 mod 0 = Constraint_Error
3 mod 1 = 0 (2#11# mod 2#1# = 2#0#)
3 mod 2 = 1 (2#11# mod 2#10# = 2#1#)
3 mod 3 = 0 (2#11# mod 2#11# = 2#0#)

0 ** 0 = 1 (2#0# ** 2#0# = 2#1#)
. . .
3 ** 3 = 3 (2#11# ** 2#11# = 2#11#)

0 and 0 = 0 (2#0# and 2#0# = 2#0#)
. . .
3 and 3 = 3 (2#11# and 2#11# = 2#11#)

0 or 0 = 0 (2#0# or 2#0# = 2#0#)
. . .
3 or 3 = 3 (2#11# or 2#11# = 2#11#)

0 xor 0 = 0 (2#0# xor 2#0# = 2#0#)
. . .
3 xor 3 = 0 (2#11# xor 2#11# = 2#0#)
```

not 0 = 3 (not 2#0# = 2#11#)
not 1 = 2 (not 2#1# = 2#10#)
not 2 = 1 (not 2#10# = 2#1#)
not 3 = 0 (not 2#11# = 2#0#)

Pred(0) = 3 (Pred(2#0#) = 2#11#)
Pred(1) = 0 (Pred(2#1#) = 2#0#)
Pred(2) = 1 (Pred(2#10#) = 2#1#)
Pred(3) = 2 (Pred(2#11#) = 2#10#)

Succ(0) = 1 (Succ(2#0#) = 2#1#)
Succ(1) = 2 (Succ(2#1#) = 2#10#)
Succ(2) = 3 (Succ(2#10#) = 2#11#)
Succ(3) = 0 (Succ(2#11#) = 2#0#)

Specific Ada 9x Issues

The revision of the language needs to address the syntax and semantics for declaring unsigned integer types in a more general fashion, similar to the current model for signed integer types. The changes ought to be upward compatible since the types would be declared in STANDARD. The following illustrates some of the possibilities when no new reserved words are used:

In STANDARD, the (minimal) required declarations are:

```
type MODULAR_nn is implementation_defined;
```

Other optional types could be supplied by the implementation.

A user-defined unsigned integer with modular arithmetic might be declared by:

```
type MY_MODULAR is mod M;
```

where M is the modulus of the arithmetic and MY_MODULAR'LAST is M - 1.

A portable syntax for declaring the largest possible distinct unsigned types would be available if the following are declared:

```
type LARGEST_MODULAR_TYPE is new implementation_defined_modular_type;
```

(These could be defined more directly if true type renaming were allowed in the revision:

```
type LARGEST_MODULAR_TYPE renames implementation_defined_modular_type;)
```

Then the following declaration may be made:

```
type MY_MODULAR is new LARGEST_MODULAR_TYPE;
```

As an alternative, if SYSTEM contains:

MAX_MODULUS : constant implementation_defined;

then the following declaration is possible:

type MODULAR **is mod** SYSTEM.MAX_MODULUS;

SELECTOR, TYPE MARK AND CONVERSIONS, ATTRIBUTE, ENUMERATION MEMBER, FUNCTIONS RETURNING MATRIX ELEMENTS, AND RECORD ARRAY MATRIX ELEMENTS COMPONENTS ALL APPEAR TO BE FUNCTIONS IN THE SOURCE

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.5.5, 4.1.4 #1, 4.3.1 #6, 4.6 #10..20, 4.1.1#5, 6.1

PROBLEM:

The notation in Ada is such that too many things resemble a function call or an array. Therefore, it is not obvious to determine what the program is expressing. The notation could be more consistent, and more elegant, by cutting down on the number of objects that appear to be functions or array elements. An improvement certainly is in the direction of improving the maintainability of Ada programs.

Several confusing situations can be created due to overuse of the "function" or "array" notation. For the first example, a type conversion would be better shown as something like:

ONE:= INTEGER'TWO; rather than ONE:= INTEGER(TWO);

The former stands out as user defined conversion is taking place. This should be consistent as attributes are treated as function calls. If confusion exists with attribute notation, then chose another symbol rather than '.

Next, the notation of Z:=F(A)(3,2) is hardly clear whether it is a function returning a matrix element or a component of a record. The latter would be better expressed with component selector notation like F.A (3,2). A discriminated record should only appear in the declaration where the object is allocated and not resemble either the function or record component case.

Thirdly, a Selector should always indicate a member of or an element of. Therefore, to be consistent, an enumeration selector could use the selector notation, ".". Instead of writing COLOR'RED the user would write COLOR.RED.

Even in the LRM example for an aggregate in 4.3.1 #6, there are not enough parentheses to make the outcome of the statement clear the attributes. The user would expect that the aggregate would initialize an element for a doubly linked list with PRED pointing to the last one and SUCC to the null one. Therefore, attributes would similarly be designated for other uses of ' to indicate a characteristic of the element. Then, for example, an array bound can be shown as VECTOR'FIRST or VECTOR'LAST. The notation for ' is just not clear.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Programming standards, renames, avoiding use of predefined attributes.

POSSIBLE SOLUTIONS:

Rework the definition of all objects that take on the appearance of

```
X:= F(A);
```

so that a function or an array is distinguishable from attributes, conversions, and enumeration members.

RANGE ATTRIBUTE FOR DISCRETE TYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.5.5

PROBLEM:

The standard defines a 'RANGE attribute for array types, but not for discrete types. This forces programmers to adjust their thinking to know when they can use a 'RANGE attribute and when they cannot.

IMPORTANCE:

IMPORTANT

This revision request is motivated primarily by concerns about symmetry and aesthetics, but the concerns are valid since there does not seem to be any good reason why such an attribute is not already defined by the existing standard.

CURRENT WORKAROUNDS:

Write loop constraints on discrete types either using 'FIRST and 'LAST attributes or the name of the type by itself.

POSSIBLE SOLUTIONS:

Add a 'RANGE attribute for discrete types.

COMPATIBILITY:

The proposed solution is quasi-compatible. All previously-compiled code will re-compile successfully unless it contains an implementation-defined 'RANGE attribute for discrete types (this is not, however, all that likely).

LEADING SPACE IN THE 'IMAGE ATTRIBUTE FOR INTEGER TYPES

DATE: October 24, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail : madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 3.5.5(10)

PROBLEM:

The leading space character in the 'Image attribute for integer types is undesirable in many cases, and it is much easier to add a leading space than to remove one.
Furthermore, it seems inconsistent with the definition of 'Image for enumeration types, where it returns only the significant characters.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Change LRM 3.5.5(10).

DOING MATH IN ADA

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 3.5.6...3.5.8, 3.5.10, 3.5.7

PROBLEM:

Model/safe number for type REAL drives the implementation into senseless conversions and becomes quite inefficient in trying to "emulate" math types not supported by the hardware. Exceptions are often forced when they are unwanted. Extra work has to be done to suppress exceptions for math types like fixed point.

The particular math model does not fit well with the IEEE floating point standard which more correctly models mathematics. Proposed approaches to incorporate a secondary standard for the IEEE package still errs in the direction of excessive code for exception handling where the hardware and interrupt handlers are sufficient. The embedded applications can perform the processing for accommodating a diverging numerical algorithm.

Also, because of the semantics on real numbers, the execution precision can be badly impacted so as not to function in realtime. By not matching machine characteristics, precision/accuracy and size cause great inefficiencies. In summary, the Ada math model does not guarantee portability. The programmer cannot tell if some range representations will be single or double precision with RANGE and DIGITS/DELTA. To the view of programmers and the machine representation, the basic unit of size is binary and not digits or delta. Further, Ada hides roundoff and truncation rules from the application which adversely affects numerical results.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Programming standards or assembler code for controlling roundoff.

POSSIBLE SOLUTIONS:

1. Delete the verbage having to do with model and safe numbers as no one is sure what the LRM is talking about and the machine model assumption meets no architecture model that we have. When porting software, it is still incumbent on the software designer to assure that the algorithms converge and are accurate on a new target architecture. The portability task can only default to the compiler when performance is not a concern.

   <<minor changes>>

2. Delete following subpara.:

   # 3 and #4, the first and last sentences of #5 in 3.5.6; #6, #7, "which has....than Float" in sentence 1 and all of last 2 sentences of #8, #9 of 3.5.7; # 8...13, sentence 2 of #14, and the notes/examples in #17...19 of 3.5.8; word model in #4, all #5, #6, #7, model in #10, notes #16 in 3.5.9; # 4, 5, 10..12, and notes #16 of 3.5.10; #1...9 and "model" in #10 in 4.5.7;

   as having no clear definition with too many antecedents for "these". If this is to remain then it must be translated to a discussion of precision/accuracy.

   <<moderate change>>

3. There is no control on truncation and rounding except in the representation section. Conversions to integer from real is defined to round rather than the customary truncate. Here, the programmer should be in control. Rounding will interfere with the numeric precision of the algorithms.

4. Add a length specification as another choice of attributes for numeric types so as to be able to map more closely to hardware and achieve the realtime efficiency needed. Non binary representations of delta and digits are not helpful in the embedded community.

5. Remove the wording that allows the representation to only get close to the specification. The language is rich enough as it is to support a named number for a range bound. Therefore, if the programmer writes a range of -1.0..1.0 it should be inclusive so that an assignment of 1 will not cause an exception to be raised. If the programmer needs only a single precision 16 bit fixed point, then he should write the upper bound as 1-T'small. On looking at the code in a maintenance phase, there should be no hidden meanings of the range that implicitly subset the declaration.

DIGITS TO SPECIFY REAL NUMBER ACCURACY AND PRECISION AND THE ASSOCIATED TRANSPORTABILITY/ EFFICIENCY PROBLEMS (SIMILARLY FOR DELTA)

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.5.6, 3.5.7

PROBLEM:

DIGITS is necessary to define attributes of floating point numbers/ objects. However, DIGITS is defined as the greatest integer of [N*LOG (10)/LOG (2)+1] which is not an exact integer for determining the binary representation. This specification never matches the architecture capability. The binary rounding for the conversion to DIGITS would occur on every assignment. Is DIGITS 6 or DIGITS 7 single or double precision? When the economy of performance counts for memory and throughput, the precise representation is needed to avoid conversions to the DIGITS specification, i.e., the typical approach of defining digits as a binary expression is desired by most algorithms. If not available in the language, the algorithm will control it appropriately. However, the current method lacks the expressibility and economy of programming that is desired in modern languages. Further, it is error prone.

DIGITS specification is neither more or less portable than being able to precisely obtain single or double precision. The machine implementation of floating point only supports single or double precision and seldom triple. With Ada, the programmer may not get support for either except in a huge library of runtimes. Floating point algorithms do not transport that easily without careful understanding of precision and impact on convergence. DIGITS specification will not ensure the ease of portability.
The precision is the key item. Therefore, non binary specifications of DIGITS (and DELTA) are only good for binary coded decimal applications at great overhead in implementation and not suitable for floating point. The embedded processing world should not have to carry the burden of inefficient language representations for programs that may not port well. With object oriented programming, the methods for porting software should be greatly improved and large re-writes should not be necessary just to tune the floating point to the actual hardware precision.

IMPORTANCE: IMPORTANT

Non binary specification of DIGITS (also DELTA) doesn't map to embedded computer floating point (fixed point) very well. In addition, many of the newer applications use the IEEE floating point standard which is one of the most impressive means of porting applications and doing high precision mathematical algorithms and Ada has no support package nor means for supporting the IEEE constructs, e.g., INFINITY, NAN, suppressing overflows automatically, in the manner intended by the IEEE floating point design considerations.

CURRENT WORKAROUNDS:

None without special math packs built into the compiler SYSTEM package.

POSSIBLE SOLUTIONS:

Provide better semantics for the representation and expression of floating point precision (also fixed point).

IMPLICIT RAISING OF EXCEPTIONS FOR INTERMEDIATE COMPUTATIONS

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.5.6, 11.6#6

PROBLEM:

Permissive semantics for raising an error exception or for a warning for intermediate results hampers the transportability of code. Without a specific requirement as to when the exception is raised makes compilers implementation dependent. The LRM should specify that the error be raised only when (1) the maximum precision of the extended representation overflows or (2) the overflow occurs on conversion to the object type on assigning the result. The control of intermediate results should not be left as an option to the implementor. The LRM should not separate the requirements in 3.5.6 and in another section far away, 11.6 #6.

IMPORTANCE: IMPORTANT

High for transportability and determination of how a compilation system will perform. (see comment on earliest detection of errors--#0-028)

CURRENT WORKAROUNDS:

Not writing complex arithmetic expressions.

POSSIBLE SOLUTIONS:

1. Provide explicit semantics for defining when intermediate results can raise an exception.

2. Allow user control to specify that greater precision can be used as long as the final results meet the demands of the expression.

SIMPLIFICATION OF NUMERICS, PARTICULARLY FLOATING POINT

DATE: October 12, 1989

NAME: B. A. Wichmann

ADDRESS: National Physical Laboratory
Teddington, Middlesex
TW11 OLW. UK

TELEPHONE: +44 1 943 6076 (direct)
+44 1 977 3222 (messages)
+44 1 977 7091 (fax)
E-mail: baw@seg.npl.co.uk

ANSI/MIL-STD-1815A REFERENCE: 3.5(6), 3.5(8), 13.7(3)

PROBLEM:

The properties of numeric data types as expressed in the RM are quite complex which leads to some confusion by programmers. For instance, there are three levels of attributes for a floating point type: those that depend upon the model number, those that depend upon the safe numbers, and those depending upon the underlying machine.

ISO/SC22/WG11 has a work item on language independent data types. It is the intention of this group to separate off numeric data types and provide a standard on this. A proposal is currently available of this entitled "Language Compatible Arithmetic Standard" (LCAS). The proposal provides a secure model for computation.

The Ada numeric model allows some insecurities. For instance, if MACHINE OVERFLOWS is false, no exception is necessarily raised when computed values lie outside the range of safe numbers. This is unsatisfactory since it is not practical to ensure algorithms are overflow free when using floating point. The LCAS guarantees that such insecurities are detected.

The Ada version of the Brown model gives rise to some flexibility which implementations do not require (such as indeterminate rounding). The LCAS avoids this, and hence is superior. Since the LCAS is language independent, it provides a standard which can be common to many programming languages, thus being easier for programmers to understand.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Use Ada 83, but ensure that MACHINE_OVERFLOWS is true for all data types used in an application. Train all programmers in the complexity of the numeric model to avoid pitfalls.

POSSIBLE SOLUTIONS:

Revise the Ada numeric facilities to take into account the LCAS. This could take a number of forms:

1. Ensure compatibility between LCAS and 9X, referencing LCAS when necessary.

2. Revise the Ada floating point model to simplify its description in the light of LCAS.

3. Revise the Ada numeric facilities significantly so that the language itself is simpler, not just the description.

It is a requirement of any standard, including Ada, to reconcile it with standards in the same area (such as the source text encoding of Ada programs and ISO 646 (ASCII)). In this case, the two relevant standards are Ada and LCAS. The 9X revision and LCAS are being done in the same time-frame. Hence the revision option 1 above would appear to be a requirement of the standardization process.

ADA DEBARS UNSIGNED INTEGERS

DATE: August 9, 1989

NAME: W B Keen

ADDRESS: Plessey Avionics
Martin Road, Havant
Hants P09 5DH
Britain

TELEPHONE: +44 794 833483

ANSI/MIL-STD-1815A REFERENCE: 3.5.4, 3.5.7

PROBLEM:

Microprocessors allow the interpretation of binary values as signed and unsigned integers, but Ada debars the latter. In our experience it is the unsigned interpretation which is more common, and the loss of this capability limits the applicability of Ada to embedded software, and its acceptance by software engineers. Associated with this, is the lack of bit-wise logical operations on integer types.

IMPORTANCE: ESSENTIAL

Ada will not be accepted as the language for embedded systems unless this glaring oversight is remedied.

CURRENT WORKAROUNDS:

Where the target architecture supports a long_integer type, limited utility is gained by type declarations of the following kind:

```
type UNSIGNED_WORD is range 0..65535;
for UNSIGNED_WORD'size use 16;
```

However the base type of this type is long_integer and all operations on objects of this type are those of long_integer. Also, no such half-measure is available for unsigned variables of the size of long_integer.

POSSIBLE SOLUTIONS:

Extension of the Ada number system to provide unsigned base types.

FLOATING POINT PRECISION

DATE: May 31, 1989

NAME: Eric C. Aker

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5212

ANSI/MIL-STD-1815A REFERENCE: 3.5.7

PROBLEM:

Ada allows you to specify precision in floating point and accuracy is promised to be to the extent of precision or better. The precision must be specified in decimal digits for Ada. LRM 3.5.7.

This is a problem because binary precision does not convert directly to decimal precision.

EXAMPLE:
In Ada an IEEE single precision floating point number with maximum precision is
type NUMBER **is digit** 6 [**range** L ..R]
6 digits gives you precision of one part in a million 1/1_000_000

The IEEE format is

sign	exponent	mantissa		
1	8	23	=	32 bits

The precision of course is a function of the mantissa only. It is one in 2**23 = 1/(2**23) = 1/8_388_608.

This lead to the situation in which the Ada compiler is allowed to round to 1/1_000_000. My specification calls for 23 bits and Ada will only promise 20 bits 1/(2*20) = 1/1_048_576.

If 7 digits is used a compiler using IEEE float forces me to use Double Long which is not what I want.

Conclusion: Ada by having a numeric specification in decimal does not allow full use of a machine dependent binary formats.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Every place that it says digits add optional [BITS].
floating_accuracy_definition ::= DIGITS [BITS] static_simple_expression
type T **is digits** [bits] d[B] [range L..R]

PERMIT 'RANGE FOR SCALAR TYPES

DATE: August 27, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne, IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 3.5(7-9), 3.6.2(7)

PROBLEM:

'RANGE is defined for array types but not for scalar types.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use 'FIRST..'LAST.

POSSIBLE SOLUTIONS:

The definition of 'RANGE for array types [3.6.2(7)] is given in terms of 'FIRST and 'LAST and actually yields the indices of the array (scalar values). It seems consistent to allow the attribute 'RANGE to apply to scalar types. This would allow, for example,

```
for INDEX in SOME_SCALAR_TYPE'RANGE loop
```

which is certainly readable.

OPEN RANGES FOR REAL TYPES

DATE: August 1, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 3.5.7, 3.5.9, 9.6

PROBLEM:

Ranges in Ada are always closed ranges (that is included their end values). There are a number of situations where a range which is open at one or both ends would more properly express the requirements of an application.

Open ranges need only apply to real types since their mathematical relevance only applies to continuous ranges and not to discrete ranges. Although an implementation of a real type will inevitably use a discrete set of values and not a continuous set, nevertheless the actual discrete set is not easily expressed. Open ranges would enable a requirement to be stated in a concise and portable manner.

An example occurs in the package CALENDAR where the range of seconds in a day is expressed as

```
subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;
```

The intent was that the value 86_400.0 should not be used, but should be expressed as 0.0 on the following day. The range should thus express 0<= x < 86,400 in mathematical terms. However, no closed upper limit can conveniently be chosen. Even

```
range 0.0 .. 86_600.0 - DURATIONS'SAFE_SMALL
```

is no good since there is no guarantee that 86_400.0 is a safe number of DURATIONS anyway (although it almost inevitably will be).

The reader will think of many other instances from his or her own application area where an open range expresses the true requirement. Note that a similar situation occurs in membership tests such as

```
x in a..b
```

IMPORTANCE: IMPORTANT

Not vital but would be nice and might be useful in tidying up various fixed point problems.

CURRENT WORKAROUNDS:

Various fiddles can be used especially if portability is not important. In many cases explicit tests will need to be inserted because the built-in range check is not correct.

Thus, if we want a <= x < b then we can write:

```
x: REAL range a .. b;
```

but if we really want to ensure that x does not take on the value b, we have to do something like

```
declare
    TEMP: REAL;
begin
    TEMP:= <new_value>;
    if TEMP = b then
        raise CONSTRAINT_ERROR;
    end;
    x:= TEMP;
end;
```

rather than simply

```
x:= <new value>;
```

POSSIBLE SOLUTIONS:

A possible notation would be to use a trailing + after a lower bound or a trailing - after an upper bound to indicate that the bound is open. For example:

```
X: REAL range a .. b-;
Y: REAL range a+.. b;
Z: REAL range a+.. b-;
```

An alternative might be to use the <and> symbols possibly replacing one of the dots so that the three lines above could be:

```
X: REAL range a..<b;
Y: REAL range a>..b;
Z: REAL range a>..<b;
```

or

```
X: REAL range a.<b;
Y: REAL range a>.b;
Z: REAL range a><b
```

REFERENCES:

See AI-00196 regarding DAY_DURATION.

FLOATING POINT NON-NUMERIC VALUES ("NAN'S")

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 3.5.7, 3.5.8, 4.5

SUMMARY:

Many hardware and software implementations of floating point arithmetic offer some representation patterns which do not correspond to numbers. These patterns can be useful in pipelined arithmetic units to avoid having to "drain the pipeline" on the occasion of certain situation--e.g. divide-by-zero, overflow or underflow. However, the Ada standard does not say how to deal with these objects, or what they mean. we do not propose any particular solution, but bring it up as an issue.

Ada is also not specific enough about the properties of floating point arithmetic, especially where certain optimizations are involved. We propose certain obvious optimizations that should always work, because they help the characterize the implementation of the floating point operations more exactly.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The Ada language should be more specific about what constitutes a floating point number, and what operations are required to be defined for them.

We recommend that the set of floating point numbers of a particular base floating point type should be capable of the following operations:

* they should be totally ordered by the "<" operation; i.e., for all x,y, exactly one of x<y, x>y, X=y is true.
* x+0.0, x-0.0 cannot fail, and x+0.0=x-0.0 =x (i.e. they have the same bit pattern).
* x-x cannot fail, and x-x=0.0 (i.e. they have the same bit pattern).
* x*1.0 cannot fail, and x*1.0 = x (i.e. they have the same bit pattern).
* x/1.0 cannot fail, and x/1.0=x (i.e. they have the same bit pattern).
* x*(1.0) = x/(-1.0) = -x (i.e. they have the same bit pattern).
* -(-x) = x (i.e. they have the same bit pattern).

CURRENT WORKAROUNDS:

If an implementation of Ada naively calls on a built-in hardware or software floating point instruction or routine, it might generate values which are not considered "numbers" by Ada. Ada should specifically disallow these values, or explain what to do with them if they are allowed.

If the set of floating-point numbers of a floating-point base type is not totally ordered, then subranges are not well-defined, and range constraints cannot be efficiently performed. Tricotomy is the law that states that in a totally ordered set of numbers, there are only three possibilities when comparing two numbers x,y--either x<y, x>y, or x=y. It should not be possible that none of the three holds between two numbers.

Certain compiler optimizations are disallowed, unless the most basic mathematical identities of arithmetic are preserved. The list given above is a minimal set.

This proposal can be thought of as an extended interpretation of Steelman requirement 3-1A and 3-1B.

NON-SUPPORT IMPACT:

Difficulties in writing portable and/or efficient code. Impossibility of performing certain types of static program analysis, including many obvious optimizations.

POSSIBLE SOLUTIONS:

The simplest solution is to disallow "not-a-numbers" (NaN's), but this would negatively impact many pipelined hardware systems.

DIFFICULTIES TO BE CONSIDERED:

Our proposal is at slight odds with the IEEE floating point proposals when it comes to comparisons, but it is more in line with the goals of the Ada language.

REFERENCES:

IEEE floating point standard

THE STATUS OF FLOATING-POINT "MINUS ZERO"

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 3.5.7, 4.5.7

SUMMARY:

The Ada standard appears to disallow the concept of "negative zero". All references to (floating point) zero appear to regard it as a single object, with no concept of sign. If a representation has a bit-pattern which corresponds to -0.0, should that bit-pattern ever appear as the result of a computation, or should it be immediately canonicalized to +0.0? If the bit-pattern corresponding to -0.0 is allowed to be manipulated, what is the result of comparing -0.0 with +0.0? We propose that an implementation function as if -0.0 did not exist; i.e. either it always canonicalizes, or arranges all operation to treat _0.0 exactly the same as if it had been +0.0.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The concept of negative zero is an algebraic abomination, and clutters up program verification routines for no apparent gain. We propose that the Ada standard tighten up the language so that no possible ambiguity exists about the non-existence of negative zero.

CURRENT WORKAROUNDS:

One of the goals of Ada is the validation and verification of software before it is delivered into an embedded system. This may require sophisticated algebraic and theorem proving techniques which are based on standard mathematical models. Mathematics has no concept of a negative zero, so allowing for the existence of such an object would complicate the job of a verification/validation tool enormously, yet only the most obscure rationale can be given for allowing the existence of a negative zero. We feel that the Ada community would be better served by not allowing the negative zero, to increase productivity of these tools.

Steelman requirement 3-1B can be interpreted as requiring compliance with this proposal.

NON-SUPPORT IMPACT:

Continuing proliferation of unneeded complications for no apparent gain.

POSSIBLE SOLUTIONS:

The two possible solutions are "continuous canonicalization", so that -0.0 never appears as a result or is ever stored into a variable, or "equivalencing", so that -0.0 is always treated exactly the same as +0.0 in all

calculations.

DIFFICULTIES TO BE CONSIDERED:

This proposal is at slight odds with the IEEE floating point proposal, but the goals for Ada are different from those of the IEEE standard.

REFERENCES:

IEEE floating point standard

SUPPRESS THE BINDING BETWEEN MANTISSA AND EXPONENT SIZE IN FLOATING POINT DECLARATIONS

DATE: October 23, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail : madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 3.5.7

PROBLEM:

The definition of the range of the exponent for model numbers of a floating point type (using 4*B, why 4 ?) creates an unnecessary binding between mantissa and exponent size.
This obliges some implementations to define types with less precision than the underlying hardware types only because of the constraint on the exponent size. For example, the D_Float floating point type on VAX processors must be defined as digits 9 just because it has a small exponent range, but its actual precision is 16 digits.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Writing non-portable programs that rely on T'Machine_Mantissa

POSSIBLE SOLUTIONS:

Separate the precision and exponent range specifications in floating point declarations, because they really are separate attributes of a floating point type. This would allow declarations such as

```
type Narrow_Exp is digits 16 exponent -30..30;
```

which should be a type with 16 digits of precision ranging at least from +1.0E-30 to +1.0E30 (the same for negative values).
The exponent part of the declaration could be made optional, using the Ada 83 rules as the default.

THE FLOATING POINT MODEL NEEDS TO BE IMPROVED

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3.5.7

PROBLEM:

To enhance robustness and portability of numeric programs, Ada incorporated the Brown model into the language definition. This was to serve as a rigorous semantic definition of floating point arithmetic for both compiler writers and numerical programmers. The Brown model, however, was originally proposed to conveniently describe all existing floating point hardware. The model is not an ideal environment for numerical programs. Consequently, the model accommodates the worst of all existing machines. More pathologically, because of the simplicity of the model, it actually admits hypothetical systems that are much worse than existing machines.

Many numerical algorithms that exploit the full capabilities of floating point computations can not be portably implemented in Ada. A detailed discussion on this problem together with some solutions is in the process of being written. [1]

> [1] "On improvement of Ada's usage of the Brown model", P.T.P.Tang, Argonne National Labs, work in progress.

We will only highlight some aspects of the problem here. Central to the problem are the two accuracy axioms found in ARM 4.5.7(4) and 4.5.7(10). One effect of these axioms is that the value of any floating point number is never known exactly. This in turn causes several anomalies in various aspects of numerical programming.

One anomaly is that numerical algorithms may not be able to simulate discontinuity closely.
For example:

```
if X <= 1.0 then
    F := expression_1;
else
    F := expression_2;
end if;
```

F may get expression_1 even when the value of X is slightly bigger than 1.0.

Another similar example using supposedly careful code

```
if X >= 0.0 then
   Y := SQRT(X);
```

```
else
    -- special action
end if;
```

unfortunately, the model does not now protect SQRT from receiving a negative argument.

Along the same lines of inaccurate comparison, is ironic that even though model numbers are special values, no portable test can determine if a value X is a model number.

Another anomaly is that the property of exact subtraction can not be supported. In short, many robust numerical programs need to exploit the fact that the statement C := A - B; will be coded to produce the exact value of A - B, whenever both are positive and the result is representable as a machine number. In the current model, such a numeric property is not even well defined when either A or B is not a model number.

Yet another anomaly is that error analysis on computations such as Y := FUNCT(X); is necessarily pessimistic whenever the slope of the function is steep. Pessimism is due to the fact that non model number X has an inherent uncertainty. On the other hand however, FUNCT would receive an unchanged machine value as input, hence reasonable implementations of FUNCT would in fact be much more accurate than the error analysis according to the model would suggest.

IMPORTANCE: IMPORTANT

It is less than desirable to have a model definition that is not as tightly specified as other parts of the language.

CURRENT WORKAROUNDS:

Do not insist on formal error analysis based on floating point model.

POSSIBLE SOLUTIONS:

Little or no changes will be required of compilers. This is close to being an administrative change.

SAFE NUMBERS FOR FLOATING POINT TYPES

DATE: September 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3.5.7(9)

PROBLEM:

According to 3.5.7(9), the safe numbers of a floating point base type "have the same number ... of mantissa digits as the model numbers of the type". This seems wrong; the safe numbers should be defined in terms of a new attribute, T'SAFE_MANTISSA, which is required to be greater than or equal to MANTISSA.

This would give an implementation the option of specifying a larger set of safe numbers than is currently possible, not only in the case where the number of mantissa bits (of the machine representation) is large relative to the number of exponent bits, but also in the case where the number of mantissa bits happens to fall between two members of the set of possible values of T'MANTISSA (i.e, the set of numbers of the form 1 + ceiling (D*log(10)/log(2)), where D is an integer in the range 1.. System.Max_Digits).

Increasing the set of safe numbers to more closely coincide with the set of machine-representable numbers will also increase the utility of the MACHINE_OVERFLOWS attribute by increasing the probability that the largest representable value will be a safe number (if the largest representable value is not a safe number, and if that value might potentially be returned as the result of an arithmetic operation, then the MACHINE_OVERFLOWS attribute must be false).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

```
Procedure example is

    X: array (1..4) of P;
    Flag: Boolean;
    procedure P4 (X: Natural) is ...;
begin
    X(1) := P4; -- Illegal, lifetime of P4 is shorter than that of type P.

    X(2) := P3; -- Illegal, parameter profile of P3 does not match that of P.

    X(3) := P2; -- Legal, but raises constraint error since the constraints on the parameter
```

```
                    subtypes of P2 do not match those of P.

        X(4) := P1; -- Legal, no constraint error.

        X(4) (17); -- A call to the current value of X(4), i.e., P1.

        Flag := F1 + F2; --Illegal, ambiguous use of "="
    end Example;
```

Remaining issues that seem straightforward, but need to be resolved, include interactions with subprogram renames, parameter passing rules for parameters of subprograms types, subtypes and derived types.

T'EPSILON IS INADEQUATE FOR REAL, FLOATING POINT NUMBERS

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.5.8

PROBLEM:

T'Epsilon is not uniform throughout the range of floating point numbers. For example, for a large negative exponent the delta is very small, e.g., the width of the mantissa times the exponent. For very large numbers, large positive exponent, the numbers are far apart. An epsilon between one number and the next for a model number [sic] is useless. T'epsilon is not applicable in the majority of algorithms requiring either fine precision or gross precision.

IMPORTANCE: IMPORTANT

Ada just does not map to any floating point machine model efficiently. The provisions are not available to support numeric applications for embedded computers in a highly productive manner.

CURRENT WORKAROUNDS:

Acquire a special math pack. Ignore the excessive built in functions for the attributes of floating point types and use the objects from the math pack.

DETERMINATION OF MANTISSAS AND EXPONENTS FOR REAL NUMBERS

DATE: September 26, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@TREES.DNET.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 3.5.8

PROBLEM:

When dealing with real numbers, there should be a defined procedure for obtaining the mantissa and exponent directly from the value of the number in an implementation-independent manner without having to carry out exponential or logarithmic calculations.

This would enable any real number to be quickly factored into two values, one of which would be an exact power of the radix and the other a number with a small absolute value. By using approximations that are good over the range of the mantissa, it is possible to carry out many numerical functions with a minimum of calculations.

As an example, imagine that the desire was for the natural logarithm of 1126.4, which is equal to 1.1*(2**10). By breaking the value into the mantissa and exponent, the value can be expressed as the natural logarithm of 1.1 plus ten times the natural logarithm of 2. This is a much easier set of values to calculate than trying to take the logarithm of the original value directly.

IMPORTANCE: IMPORTANT

I believe this feature would be a great aid in the efficiency of "number crunching" code and should require minimal effort for implementation.

CURRENT WORKAROUNDS:

The only implementation-independent means that I know of is to repeatedly divide the value by the radix repeatedly until the quotient has an absolute value less than the radix. This assumes that the absolute value of the original value was greater than one. If the absolute value was less than one, the value would be multiplied by the radix until the product was greater than one. Not only is this time consuming, it also introduces round-off error into the calculation

POSSIBLE SOLUTIONS:

The simplest solution would be to add two predefined functions to the language definition with the calling sequences

```
function EXPONENT (VALUE : in universal_real) return INTEGER;
```

function MANTISSA (VALUE : **in** universal_real) **return** universal_real;

Addition of this requirement to the language specification should be a very simple task for implementation.

OPERATIONS ON REAL NUMBERS

DATE: September 26, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@TREES.DNET.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 3.5.8

PROBLEM:

Predefined functions should be added to the language definition to support the trigonometric and exponential functions, as well as some of the other commonly used mathematical functions. In addition, a subtype of REAL should be established for angles in radians.

This would appear to be a reasonable subject for the Ada language specification since these functions are used in a wide variety of programs.

IMPORTANCE: IMPORTANT

Trigonometric and logarithmic calculations are used in a wide variety of programs. Failure to provide a common definition for the interfaces will tend to result in multiple packages being used for numeric calculation, resulting in redundant code in the applications. This will not only increase the programming effort, but will also reduce maintainability and the ability to modify the code.

CURRENT WORKAROUNDS:

The current workaround is to have these functions redefined by each organization wishing to use them in their code. Although there are some organizations offering Ada packages for mathematical functions, there is no requirement for coordination between them.

POSSIBLE SOLUTIONS:

Many of these functions are already defined in the FORTRAN language specification, and the bulk of the calling sequences could be lifted directly from this document.

Implementation would not appear to be a difficult matter since numerous algorithms for determining these functions have been published.

Even if it was decided that there was no need for the subroutines to be made mandatory, the format should be fixed so that any subroutines used for these functions will follow the same format.

ENTIER FUNCTIONS ON REAL TYPES

DATE: September 25, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3.5.8, 3.5.9

PROBLEM:

For applications requiring entier functions, there is no way to obtain an efficient solution in Ada.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use a generic function, e.g.:

```
generic
        type Real is digits <>;
        type Int is range <>;
function Entier (Value : Real) return Int;

function Entier (Value : Real) return Int is
        I : Int := Int(Value);
        R : Real := Real(I);
begin
        if Value >= 0.0 then
                        if R > Value then
                                I := I - 1;
                        end if;
        else
                        if R < Value then
                                I := I + 1;
                        end if;
        end if;
        return I;
end Entier;
```

(Note: a similar generic workaround can be provided for fixed-point types.)

This function depends on explicit conversion and is clearly less efficient than a built-in function in the language would be. In addition, this function must be instantiated tediously for each floating-point and integer type pair, in most cases producing redundant code bodies because generic bodies are not usually sharable.

POSSIBLE SOLUTIONS:

Provide an efficiently-implementable attribute for every real type:

P'Entier For a prefix P that denotes a real type:

> This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type universal_integer. The value is the usual entier function of the actual parameter.

ADDING ATTRIBUTES 'IMAGE AND 'VALUE TO FLOATING POINT TYPES

DATE: August 18, 1989

NAME: Goran Karlsson

ADDRESS: Bofors Electronics AB
Nettovagen 6
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 222 90

ANSI/MIL-STD-1815A REFERENCE: 3.5.8

PROBLEM:

Inconsistency between attributes of floating point discrete types.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Writing two functions to do the work.

POSSIBLE SOLUTIONS:

MANTISSA OF FIXED POINT TYPES UNREASONABLY SMALL

DATE: August 20, 1989

NAME: James W. McKelvey

ADDRESS: R & D Associates
P.O. Box 5158
Pasadena, CA 91107

TELEPHONE: (818) 397-7246

ANSI/MIL-STD-1815A REFERENCE: 3.5.9, 6

PROBLEM:

The LRM is too lax in it definition of B, the mantissa required for a fixed point type. Example:

type F_Type **is delta** 0.3 **range** 0 .. 1.1;

Two different Ada compilers came up with the following:

Mantissa 2 bits
Large 0.75
Small 0.25

This defines the model numbers 0.0, 0.25, 0.50, and 0.75. What happened to 1.0? The answer to be found in (3.5.9, 6), which states "the value of B is chosen as the smallest integer number for which each bound of the specified range is either a model number of lies at most small distant from a model number". So, since 1.0 is exactly small distant from 0.75, it need not be a model number, and the mantissa need only be 2. I submit that 1.0 should "obviously" be a model number, and a mantissa of three is required.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Be extremely careful in defining fixed point types.

POSSIBLE SOLUTIONS:

Change the working in (3.5.9, 6) to read "or lies less than small distant from a model number". Also, add a note after (3.5.9, 17) which states that "The range of model numbers must extend as close to the bounds of the range constraints as is possible under the definition of model numbers". Furthermore, "For a fixed-point type T, T'First and T'Last will be model numbers of T".

FIXED POINT SCALING AND PRECISION

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.5.9

PROBLEM:

The Ada concept of rational or expressions for binary representation for range and precision makes it difficult to determine the amount of precision and the scale factors for an application program. Allowing the compiler and runtimes to decide the representation is highly error prone and is not easily controllable. Most of the decimal representations are inexact therefore a binary version will be the most frequently used version. Here, a scale factor would have been more useful. Further, embedded computers use fixed point primarily for speed and accuracy. The algorithm precision is masked by the decimal representation for the specification. The programmer is forced to write an expression to compute scale factors and deltas when it would be easier to recognize in a binary format with length specifier and scaling.

IMPORTANCE: IMPORTANT

Very important to embedded applications where finer precision than floating point is needed. The compiler vendor is not able to recognize all the special cases in the expressions, e.g., 2**16-1 rather than something like sign & length (15), to determine when only shifts and masks are necessary and the binary representation is exact.

CURRENT WORKAROUNDS:

Write many expressions for well known static constants that the compilation system may or may not be able to use to any advantage.

POSSIBLE SOLUTIONS:

1. Remove the wording for model/safe numbers.

2. Provide binary expressions, such as binary length and scale, for control of DELTA/RANGE precisely.

3. Allow the compiler to support efficiently single and double precision fixed point and control scaling/masking for the more frequently used representations in embedded applications, e.g., fractional form.

DETERMINING 'SMALL FOR FIXED POINT TYPES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3.5.9

PROBLEM:

The effects of a representation clause should be transparent to a program which is written entirely in Ada (with no Machine_Code insertions) and does not use implementation-dependent features such as Unchecked_Conversion, Representation Attributes, or System.Address. Time and space requirements of the program may be affected, but otherwise the program should behave the same with or without a representation clause.

The one unfortunate exception to this rule is a length clause specifying T'Small for a fixed point type.

For example, removing the length clause from

```
        type T is delta 0.01 range 0.0 .. 1000.0;
        for T'Small use 0.01;

        X: T := 0.0;
begin
        for I in 1 .. 100 loop
        X:=X + 0.01;
        end;
        if X not in 0.9 .. 1.1 then
                raise Program_Error;
        end if;
```

will probably cause the program to fail because each iteration through the loop may then increment X by any value between 1/128 and 1/64.

One should not have to resort to representation specifications in order to get a fixed point type T such that T'Small is not a power of two.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use representation specifications.

POSSIBLE SOLUTIONS:

Given a fixed point type declaration of the form

type T **is delta** D **range** L .. R;

the implementation should be required to either select a basetype such that T'Delta is an integral multiple of T'Base'Small or reject the declaration of T. T'Small should be defined to be equal to T'Delta.

Given a fixed point constraint occurring in a subtype indication, such as

subtype S **is** T **delta** D2 **range** L2 .. R2;

S'Small should be defined as the largest integral multiple of T'Delta that is less than or equal to D2.

Note that these two rules preserve the equivalence given in 3.5.9(8-9).

UNIFORM REPRESENTATION OF FIXED POINT PRECISION FOR ALL RANGES

DATE: November 2, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783

ANSI/MIL-STD-1815A REFERENCE: 3.5.9, 13.2(11 & 12), 13.7.1(6)

PROBLEM:

Paragraph 3.5.9(4) states: "A canonical form is defined for any fixed point model number other than zero. In this form: sign is either +1 or -1; mantissa is a positive (nonzero) integer; and any model number is a multiple of a certain positive real number called small, as follows:

sign * mantissa * small

Paragraph 3.5.9(6) states: "For a fixed point constraint that includes a range constraint, the model numbers comprise zero and all multiples of small whose mantissa can be expressed using exactly B binary digits ...."

Paragraph 13.7.1(6) defines System.Fine_Delta as the smallest delta allowed in a fixed point constraint that has the range constraint -1.0 .. 1.0.

These paragraphs prevent the representation of an asbtraction that involves are range of values such as 100.0 .. 102.0 as a fixed point type with the same precession as the range -1.0 .. 1.0. For example:

```
type Sensor_Type is delta System.Fine_Delta range 100.0 ..102.0;
```

In many situations, a required precision will not be supported for a range of required values because values must be represented as just a multiple of small.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Define an abstract type such as the following:

```
with System;
package Sensor
is
        --
        -- Define types to support a Sensor Value range of100.0 .. 100.1
        -- which is accurate to the Maximum Accuracy of the System
        --
```

```
        type Value_Range_Type is delta System.Fine_Delta range 0.0 .. 1.0;
        type Value_Offset_Type is range -100 .. 100;

        -- Base Offset of physical sensor
        Base_Offset: constant Value_Offset_Type        := 100;

        type Value_Type is private;

        -- Define Identity values for math operations
        Zero: constant Value_Type;
        One: constant Value_Type;

        -- Define math operations, relational operations, image
        -- operation, etc.

private

        type Sensor_Type is
                record
                        Base_Value:     Value_Range_Type;
                        Offset:  Offset_Type;
        end record;

end Sensor;
```

POSSIBLE SOLUTIONS:

Redefine the canocal form of a fixed point type be:

sign * mantissa * small + offset

Make the necessary adjustments to the fixed point operators (e.g., for the Scale_Type defined above, given that Scale_Type'Small = 0.1, then it must be true that 100.0 + 0.1 = 100.1).

Add a representation specification for the offset of a fixed point type.

DECIMAL

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 3.5.9(1-19), 13.2(11-12)

PROBLEM:

Ada does not handle packed decimal arithmetic properly as does PL/1 and Cobol. Two requirements are necessary for packed decimal arithmetic: a decimal delta and a large range. This is a serious deficiency in the language. PL/1 has solved this problem with binary and decimal attributes: binary(31,0) is integer; binary(31,5) is delta 0.03125 in Ada. However, Ada has nothing which corresponds to decimal(9,2). There is no need to distinguish between binary(31,0) and decimal(9,0)

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

None that are attractive. Long_float could possibly be used, but the size of long_float is not well defined and also leads to rounding errors. A multiprecision package could be written, with type Money is private, and represented as an array of integers, but multiprecision arithmetic is extremely difficult when the language provides no access to the carry bit or to a double_length product of two integers. Also, there is no way to specify a literal of a private type.

POSSIBLE SOLUTIONS:

Consider the following declarations:

```
type Money is delta 0.01 range -999_999_999_999_999.99
                            -- +999_999_999_999_999.99;

for Money'small use 0.01;
```

Firstly, the representation clause (at least for the DEC/VMS Ada compiler) generates an error stating that 0.01 is not a power of 2. If we omit the clause, then 1/128 will be used, in which case $1.00 = $0.78 due to rounding error. Secondly, the type declaration produces an error stating that no predefined type is able to satisfy these requirements. This is correct, but money measured in trillions of dollars is not completely unreasonable in an application. One compiler, for example, will limit the range to -21_474_836.48..+21_474_836.47: $21 million is definitely inadequate.

There are two amendments required of the language before this can be implemented properly: decimal deltas, and large ranges for fixed point arithmetic.

Decimal deltas:

1. The language should be amended to allow the clause:
   for Money'small **use** 0.01;
   with a non power of 2 small. All compilers should be required to support this. The internal representation should simply be an integer counting the number of 0.01 units (pennies).

2. Leave the small as currently implemented and add a new reserved word: DECIMAL. Then one would say:
   type Money **is** decimal 0.01 **range** -999_999_999_999_999.99
   .. +999_999_999_999_999.99;

3. Introduce a new pragma into the language:
   pragma decimal(Money);
   would require the compiler to use a 'small which is a power of 10 and not of 2. It would suggest to the compiler that packed decimal arithmetic should be used, if available.

Large ranges:

4. All compilers should be required to implement double precision integer arithmetic for both binary and decimal smalls. This can be done by multiple precision arithmetic routines, which, although not efficient can do the job.

5. Alternatively: all compilers should have shipped with them a package Multiprecise_Arithmetic which uses arrays to represent multiple precision numbers with either decimal or binary points and which provides all the usual arithmetic operators. It should probably be generic. Also, the language would need a method of specifying literals for the private type to be exported by this package.

Recommendation:

Adding a new keyword to the language will inevitably cause some programs to fail and is not recommended so much as other options.

The changes should be: Add pragma decimal as required to be implemented. In this case the 'small shall be required to be a power of 2 instead of 10, making existing programs less likely to break. Require that all implementations furnish decimal numbers of at least 18 digits or 63 bits. Do not require that packed decimal be used, as long as multiple precision binary is available, although inefficient.

All existing packages (including TEXT_IO) should provide all features to the decimal types that are now provided to the fixed types.

Objective:

Ada should be able to do anything PL/1 or Cobol can do, and as easily.

FIXED POINT MODEL NUMBERS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3.5.9(6)

PROBLEM:

In 3.5.9(6), the text "at most" should be replaced with "less than".

In this example,

```
type T is delta (2.0 ** (-8)) range 0.0 .. 1.0;
```

this would mean that T'Mantissa would be 9, not 8, and that therefore 1.0 would be a model number of T (currently, it is not). Currently, an implementation has the option of using an 8-bit representation for objects of type T despite the fact that the user clearly desires that the type T include at least 257 distinct values.

If a bound specified in a fixed point constraint is an integral multiple of the associated Small value, it seems absurd for that bound to be excluded from the set of model numbers defined by the fixed point constraint.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

MULTI-DIMENSIONAL ARRAY STORAGE

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 3.6

PROBLEM:

The way array components are associated with allocated memory resources is not defined in the language. For Numerically Intensive Computing (NIC) applications it is very important that this storage mode is known. The essential information is:

- for increasing index the values of the addresses where the components are stored should change monotonically,
- for multi-dimensional array objects the storage mode is row major or column major or otherwise:
  a) for all multi-dimensional arrays the first index runs the fastest, then the next, etc. (column major),
  b) for all multi-dimensional arrays the last index runs the fastest, then the last but one, etc. (row major),
  c) other storage methods (like for sparse matrices, or block storage),
- where there is an inefficiency difference in the way indexing is implemented, all compilers should do this in the most efficient way.

We need this information because many linear algebra algorithms have both row-oriented and column-oriented variants, and the efficiency of these variants depends strongly on the storage mode used. It is urgently required that this information can be obtained, and it cannot be obtained now.

Additionally, the user also has no (implementation-independent) option to choose a preferred storage mode. It can be imagined that some Ada implementations would allow both row major and column major matrix storage and leave the choice to the user. We assume that in such cases it is not feasible to have the possibility of choosing the storage mode for EVERY individual matrix object, but rather for all multi-dimensional array objects collectively. Then, if all matrices are stored the same way dead-code elimination would be possible.

We do not want to require this additional facility, but only wish mention it here as a related issue that would be nice to have it available.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

The current approach is implementation-dependent.

POSSIBLE SOLUTIONS:

Preferably by an attribute for obtaining the information about the actual storage mode, or else through the package SYSTEM. A type for the attribute could be:

```
type MATRIX_STORAGE_MODE is (ROW_MAJOR, COLUMN_MAJOR, OTHER);
```

The additionally mentioned control over the storage mode could be given through a representation clause for setting the attribute to one of its possible values.

UNIVERSAL EXPRESSIONS IN DISCRETE RANGES

DATE: January 26, 1989

NAME: R. David Pogge

ADDRESS: Naval Weapons Center
EWTES - Code 6441
China Lake, CA 93555

TELEPHONE: (619) 939-3571
Autovon: 437-3571
E-mail: POGGE@NWC.NAVY.MIL

ANSI/MIL-STD-1815A REFERENCE: 3.6.1 paragraph 2

PROBLEM:

Ada allows range constraints with universal_integer bounds, but not with universal_expression bounds. That is,

```
for i in   0..5 loop -- legal
for i in  -5..5 loop -- illegal
```

IMPORTANCE: ADMINISTRATIVE

This is ADMINISTRATIVE because it makes the language more consistent and easier to learn.

CURRENT WORKAROUNDS:

Explicitly type the loop index.

```
for i in integer range -5..5 loop
```

POSSIBLE SOLUTIONS:

Insert the words "universal_expression" in 3.6.1 (2) so it reads,

> "... if each bound is either a numeric literal, universal_expression, a named number..."

OBTAIN CONSTRAINTS FROM A VARIABLE'S INITIAL VALUE

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3.6.1(6), 3.7.2(8)

PROBLEM:

It would be useful if one could declare a variable of an unconstrained array type (or of a discriminated type whose discriminant components lack default initial values) and obtain the variable's constraints from the (explicitly specified) initial value of the variable.
For example, the following should be legal:

```
type No_Default_discriminant_Value (D:Natural) is
        record
                F:String (1.. D);
        end record;
type Unconstrained_Array is array (integer range <>) of
Integer;

function Foo return Unconstrained_Array is ...;
function Bar return No_Default_Discriminant_Value is...;
package P is
        Foo_Var : Unconstrained_Array := Foo;
        Bar_Var : No_Default_Discriminant_Value := Bar;
        -- note that Bar_Var'Constrained = True
end P;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

One frequently sees the following sort of workaround:

```
        X : constant String := Other_Package.Some_Function;
        Y : String (X'Range) :=X;
```

This is awkward and likely to be inefficient.

POSSIBLE SOLUTIONS:

FUNCTIONS, UNCONSTRAINED TYPES, AND MULTIPLE RETURN VALUES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.6.1(6), 3.7.2(8), 6.5(1)

PROBLEM:

When working with unconstrained types, functions are vastly superior to procedures, because their return values can be used to provide the bounds for constants of the type:

```
declare
        Some_Constant : constant Some_Unconstrained_Type :=
                Some_Function_Returning_Unconstrained_Type;
begin
        [process]
end;
```

Unfortunately, it is sometimes the case that a function returning an unconstrained type needs to also return a status of some kind, and this is not well-supported by the current standard.

IMPORTANCE: ESSENTIAL

This problem with the language has no clean workaround.

CURRENT WORKAROUNDS:

Parameters to functions can be passed by reference (by passing a pointer to the actual parameter and aliasing it internally as a local variable); since the pointer is not itself modified, this compiles. Unfortunately, this is an awful thing to have to do, and a cleaner solution should be provided by the standard.

Alternatively, procedures can be used in place of functions, but this requires the programmer to guess ahead of time how big to make a constrained buffer for the unconstrained type (this is necessary, for example, in order to use the current Text_Io.Get_Line procedure); the actual bounds then have to be maintained by the programmer.

Another strategy is to declare a record with a constraint:

```
type results (Size :Integer) is
        record
                Unconstrained_Field : Some_Unconstrained_Type
                                        (1..Size);
                Status : Some_Status_Type;
        end record;

function Some_Function return Results;

Bar : Results := Some_Function;
```

POSSIBLE SOLUTIONS:

Several different approaches have been suggested:

* Permit parameters of mode OUT and IN OUT on functions. This would allow the problem described above to be solved as follows:

```
function Returns_Unconstrained_Type
                                (And_This_Status : out Status_Type)
                                        return Unconstrained_Type is ...
```

There are, however, lots of drawbacks to this solution, such as introducing side-effects and distorting commutativity.

* Extend the syntax of the language to allow multiple values:[3]

```
function Some_Function return Some_Unconstrained_Type, Some_Status_Type;

declare
        X : constant Some_Unconstrained_Type, Y :    Some_Status_Type := Some_Function;
begin
        [process]
end;
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

[3]This is just a sketch of the possibility, not a complete fleshing out of a suggested syntax. It might be possible to make this work with both positional and named association, given more thought.

PROVIDE A UNIFICATION OF CONSTRAINED AND UNCONSTRAINED ARRAYS

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3.6(2), 12.1(2)

PROBLEM:

There is a significant problem for users that write seemingly reasonable programs, have them "completed", then find a minor addition is needed. The problem is most severe when the minor addition can be accomplished by instantiating a generic library package. (Applicable to numerical packages, graphical packages, data base binding packages and many others.)

The other view of the problem is that of the generic library package writer: How can the generic library package be written to be most usable as an "after the fact nice add on?"

Consider the following fragment of a users working application:

```
package arrays is -- used by an average user
        type P is digits 6;
        type D is (North, East, Vertical);
        type A_0 is array(D, D) of P; -- all dimensions constrained OK
--      type A_1 is array(D, POSITIVE range <>) of P; -- can not mix
--      type A_2 is array(POSITIVE range <>, D) of P; -- can not mix type A_3 is array(POSITIVE range
        <>, NATURAL range <>) of P; -- OK, but
        type A_4 is array(D, NATURAL range 0..2) of P; -- both constrained
        type A_5 is array(1..3, D'first..D'last) of P; -- both constrained
end arrays;
```

Now, a library package that is to have broad application:

```
generic -- a useful addition that is later discovered in a library type P is private;
        type D is (<>);
        type A_0 is array(D, D) of P; -- OK, absolutely unique
--      type A_1 is array(D, POSITIVE range <>) of P; -- can not mix
--      type A_2 is array(POSITIVE range <>, D) of P; -- can not mix type A_3 is array(INTEGER range
        <>, INTEGER range <>) of P; -- too general
        type A_4 is array(D, NATURAL) of P; -- still wrong constraint
        type A_5 is array(INTEGER, D) of P; -- will not take encompassing type
package X is
end X;
```

The user tries to augment the application with an instantiation:

```
with X;
with ARRAYS; use ARRAYS;
procedure Y is -- bound to get many constraint errors
        package XX is new X( P      => P,
                             D      => D,
                             A_0    => A_0,
                             A_3    => A_3, --     must be specifically POSITIVE
                             A_4    => A_4, --     must be new subtype of NATURAL
                             A_5    => A_5);--     want constrained universal integer
begin
        null;
end Y; --       this compiles and goes into execution, then CONSTRAINT_ERROR !
```

There is a possible unification that needs to be considered.

IMPORTANCE: IMPORTANT

As more Ada applications are "completed" there is an increasing need to add features without rewriting the applications. The number of useful library packages is growing. The general concept of reuse is being implemented. Thus, there must be a broadening of the utility of generic units.

CURRENT WORKAROUNDS:

Either the application is modified or the library package is modified. There is minimal practical ability to write generic packages that can placed in a library and used by existing applications.

POSSIBLE SOLUTIONS:

Just one incomplete idea.

1) An array index position may be constrained (given lower and upper bounds) or may be unconstrained (given <>). When an object of an array type is declared, the constrained index positions must not be given bounds and the unconstrained positions must be given bounds.

2) If a type mark is used where the syntax of a range is needed, the range is type_mark'FIRST..type_mark'LAST

3) A generic actual parameter must be a subtype of the generic actual parameter. The generic actual parameters constraints are used throughout the generic unit.

4) A generic formal index position that is unconstrained will match a generic actual index position, using the generic actual index position constraints as subscript constraints for that position. Such an index position must be given bounds when an object is declared in the generic unit. The attribute index_type_mark'POSITION_RANGE is available to specify the widest possible bounds.

VARIABLE-LENGTH STRING

DATE: August 11, 1989

NAME: Larry Langdon

ADDRESS: Census Bureau
Room 1377-3
Federal Office Bldg 3
Washington, DC 20233

TELEPHONE: (301) 763-4650
E-mail(temporary): langdonl@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 3.6.3

PROBLEM:

Strings are inadequate in Ada. It is very frequently the case that the length of a string is not known until it is formed...after it has been declared. This leads to ugly, clumsy constructions (blank pad everything, keep track of length separately, play tricks with DECLARE's and constant strings, etc.). The obvious solution of writing a variable-length string package (see LRM, section 7.6) is unsatisfactory: you are lead to a limited private type because neither the standard equality test nor assignment are appropriate. (you want the both to ignore everything beyond the actual length of the strings) For limited private types, however, you have no assignment statement at all. We implemented such a package and found that using a procedure (SET) for assignment was error-prone and hard-to-read. This even for experienced programmers and even after getting beyond the initial learning curve for the package.

IMPORTANCE: IMPORTANT

As mentioned above, we have found that the present alternatives beget error-prone and less readable code.

CURRENT WORKAROUNDS:

a) Declare your string to be the maximum length needed and pad it at all times with blanks, NUL's, or whatever. This only works when you have some character known to be insignificant at the tail of the string. It can be very inefficient when there is a wide range of actual string lengths.

b) Declare your string to be the maximum length needed and have a separate integer variable to keep the actual length of the string. This causes extra coding which represents additional error possibilities and reduces readability.

c) Use a package such as that in section 7.6. As mentioned above, the lack of an assignment statement interferes with both reliability and readability.

POSSIBLE SOLUTIONS:

a) Define a variable string type within the language.

b) Allow overloading of assignment (:=), thereby permitting a package such as that in LRM section 7.6 to be properly constructed.

RESTRICTING "STRING" TO CHARACTER

DATE: May 21, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.6.3

PROBLEM:

It seems overly restrictive to only allow CHARACTER as string elements and not allow BOOLEAN or BYTE strings. Typically, interface data can be generated and received as strings of known length. The structure of the language easily supports an extension by providing for BOOLEAN or BYTE to follow the "of" in the object declaration.

IMPORTANCE: IMPORTANT

Embedded systems seldom have the same character representation or interface needs for strings.

CURRENT WORKAROUNDS:

Treat as arrays and hope that the packing can be specified in rep specs.

POSSIBLE SOLUTIONS:

Allow other constructs/objects.

VARYING STRINGS

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 3.6.3

PROBLEM:

Ada has no mechanisms for defining and manipulating varying strings.

IMPORTANCE: IMPORTANT

Without this capability, Ada will be avoided for applications for which character handling is important. Where use of Ada is mandated, reliability will suffer. Solutions involving vendor-supplied packages will not be transferrable across implementations, and will cause a skills-transfer problem.

CURRENT WORKAROUNDS:

Use unconstrained strings. This can be extremely expensive, and in any event has the wrong semantics.

Define varying_string as a type of package. However, limitations on the overloading of assignment and equality operations make this unacceptable. Further, this makes it awkward to treat fixed and varying strings symmetrically.

POSSIBLE SOLUTIONS:

Change 3.3.2(2) to read

subtype_indication ::= [VARYING] type_mark [constraint]

with a paragraph specifying that varying can only be used for an array type.

Add to 5.2.1(1)

If the array variable is a varying array, then the size of the expression becomes the current size of the array variable; the exception CONSTRAINT_ERROR is raised if this exceeds the declared size of the array variable.

Add to 11.1(5)

the current length of a varying array,

In addition, there should be attributes, functions and procedures to append, truncate and determine maximum length.

Alternatively, the above capability could be limited to strings.

RECORD TYPE

DATE: October 10, 1989

NAME: William Thomas Wolfe

ADDRESS: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: (803) 656-2847
E-mail : wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 3.7

PROBLEM:

The record type, which is a composition mechanism, does not automatically compose all operations which are available to each and every component into an identically-named operation over the resulting record type.

IMPORTANCE:

CONSEQUENCES:

Operations such as "=" and Assign, as well as any other operations which might reasonably be expected to compose automatically, must be composed manually, leading to a loss of programmer productivity.

CURRENT WORKAROUNDS:

As described in CONSEQUENCES.

POSSIBLE SOLUTIONS:

Introduce a new pragma ADT (type); which can be used on a limited private type defined within a package specification; this indicates that the type and the associated operations within the package specification constitute a user-defined data type, and would instruct the compiler to automatically compose operations which are available on each and every component of a given record type into equivalent operations over that record type.

Alternatively, perhaps there could be some means of specifying that only specific operations over a user-defined data type are composable (e.g., pragma Composable); the point is that there needs to be *some* mechanism by which general composition can be done automatically. Making only specific operations composable (such as assignment and equality) rather than arbitrary operations should NOT be considered a reasonable solution to the problem.

PROBLEMS REGARDING ANONYMOUS ARRAY TYPES

DATE: October 11, 1989

NAME: J G P Barnes (endorsed by Ada (UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 3.7

PROBLEM:

It is generally very irritating not being able to use anonymous arrays in records. Extra type names have to be introduced which are only used once; this causes clutter, obscures the situation and may introduce a false sense of abstraction. Thus instead of neatly writing

```
type STACK(MAX: NATURAL) is
        record
                S: array (1 ..MAX) of INTEGER;
                TOP: INTEGER;
        end record;
```

we have to laboriously write

```
type INTEGER_VECTOR is array (INTEGER range<>) of INTEGER;
type STACK(MAX: NATURAL) is
        record
                S: INTEGER_VECTOR(1 ..MAX);
                TOP: INTEGER;
        end record;
```

where the intermediate type INTEGER_VECTOR may well not be used elsewhere at all

Note that anonymous array types were allowed in 1980 Ada and so this irritation did not then arise.

As a further and more severe example consider the problem of creating two dimensional array structures; there is a choice between declaring a genuine two dimensional array or alternatively an array of arrays. Each has its merits in different circumstances.

Thus we might have

```
type MATRIX is array (INTEGER range <>, INTEGER range <>) of REAL;

type VECTOR is array (INTEGER range <>) of REAL;
```

To create an array of arrays, we need to give a name to the intermediate type. And moreover the component subtype must be constrained so that the number of columns must be pinned down first. So

```
type MATRIX_3_6 is array (1..3) of VECTOR(1 .. 6);
```

We can now declare a two dimensional array or an array of arrays

```
TDA: MATRIX 1..3, 1 .. 6);
AOA: MATRIX_3_6;
```

We could of course have declared TDA without introducing a name for the type MATRIX at all. But on the other hand we could not have avoided giving a name to the intermediate type in the case of AOA; the most we could have done is write

```
ADA: array (1.. 3) of VECTOR(1 .. 6);
```

A curious anomaly now occurs in the case of discriminated records; We can declare

```
type RECTANGLE (ROWS, COLUMNS: POSITIVE) is
        record
                MAT: MATRIX(1 .. ROWS< 1 .. COLUMNS);
        end record;
```

where the internal structure is a two dimensional array but we cannot create the analogous record where the internal structure is an array of arrays. We want to say something like

```
type RECTANGLE (ROWS, COLUMNS: POSITIVE) is
        record
                MAT: array (1..ROWS)of VECTOR(1 .. COLUMNS);
        end record;
```

but we cannot because an anonymous array type is not allowed in records. Moreover we cannot name the type outside the record because the discriminant COLUMNS would not be visible.

Unlike the first example there is no workaround at all in this case.

The inability to choose the representation is annoying when one considers that we can declare the private type with discriminants

```
type RECTANGLE (ROWS, COLUMNS: POSITIVE) is private;
```

and might then hope to provide either of the above formulations as the full type.

Again this problem did not arise in 1980 Ada.

IMPORTANCE: IMPORTANT

Not very important but it illustrates the sort of frustration that Ada causes because of its nonorthogonality. The Ada type model brings surprises which may be symptomatic of a lack of consistency.

CURRENT WORKAROUNDS:

As illustrated above.

POSSIBLE SOLUTIONS:

The language could revert to the situation in 1989 where a record component could be declared with an array type definition. But there is clearly a need to investigate why it was changed in the first place. There were a number of changes from 1980 that restricted the user in a similar manner (e.g., requiring type mark rather than subtype indication in many situations); this one is the only one that is a real nuisance.

ANONYMOUS POINTER TYPES

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 3.7

PROBLEM:

In constructing lists, trees, and other graphs, it is necessary to have a record containing a pointer to records of the same type. This is an extremely common situation. The language requires that the record type be declared before it is referenced in the access type definition, and then fully defined after the access type definition:

```
type Egg;
type Chicken is access Egg;
type Egg is record
        Yolk: chicken;
end record;
```

Not only are there three type declarations (syntactically) to define what is conceptually one type, but there are two names to describe one thing. We will only speak of chickens in the body of the program. The eggs are only needed in a few formalisms. We do, however, need to talk about eggs in creating a deallocator:

```
procedure Dispose is
        new Unchecked_Deallocation (egg,chicken);
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

See the problem section.

POSSIBLE SOLUTIONS:

Permit the subtype indicator in the component declaration of a record declaration (or maybe anywhere) to be of the form "access<subtype indicator>".

```
type Chicken is record
        Yolk: access chicken;
end record;
procedure Dispose is
```

```
new unchecked_deallocation(chicken);
```

Of course, it can be argued that this violates principles of sound software engineering, since it creates an anonymous type, but the formalities behind numeric types and array types produce multitudes of anonymous types anyway.

There is also a problem with Dispose. If the argument to dispose is the pointer type, then it is an anonymous type, and there might be more than one such anonymous type, which is a violation of type uniqueness. If the argument is the type of the item denoted, then Dispose cannot have the side effect of setting the pointer to null. I am of the opinion that this side effect is of little value.

ASSIGNMENT TO A DISCRIMINANT

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 3.7.1(9)

PROBLEM:

For large discriminated records, a complete object assignment just to change the discriminant is frequently very cumbersome. Many times the component values must be determined in a sequence of processing thereby making it awkward to have all component values available at a given point in the execution.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

No workarounds exist.

POSSIBLE SOLUTIONS:

Delete 3.7.1(9) in order to allow discriminant assignment to be on par with component assignment. Note that 4.1.3(8) will still be in effect.

DIRECT ASSIGNMENT TO DISCRIMINANTS IS NOT ALLOWED

DATE: July 12, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10 - Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 3.7.1(9)

PROBLEM:

Direct Assignment to Discriminants is not allowed.

The problem with this rule is that on the one hand, it is not really necessary (it is a protection against undefined values in a specific context), and on the other hand it is the only rule that prevents support for mutants of limited types.

IMPORTANCE: ADMINISTRATIVE

Administrative as a single instance of the problem. The general approach of typing to half-protect against undefined values in Important.

CURRENT WORKAROUNDS:

Don't use limited mutants, put all components of the variant part at one level, verify consistency yourself.

POSSIBLE SOLUTIONS:

One should define components that depend on discriminants undefined immediately after assignment to the discriminant, such that programs that try to fiddle variants are erroneous because of undefined value access.

High quality compilers should include a general (suppressible) verification for undefined values, or should at least initialize the undefined dependent components to a known value in order to prevent programs to try using mutants as type conversion machines.

PARTIALLY CONSTRAINED DISCRIMINATED SUBTYPES

DATE: October 21, 1989

NAME: Gary Dismukes

ADDRESS: TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9891

TELEPHONE: (619) 457-2700 x322

ANSI/MIL-STD-1815A REFERENCE: 3.7.2

PROBLEM:

Ada's existing mechanism for constraining discriminated types is too restrictive. While range constraints can be used to constrain scalar subtypes to a range of values, discriminant constraints can only be used to fully constrain the values for a given subtype so that all values of the subtype have discriminants fixed to a single value.

IMPORTANCE: IMPORTANT

The addition of a feature for expressing partially constrained discriminated subtypes would provide much greater flexibility in the use of discriminated types and enable programmers to express logical restrictions on values of a discriminated variable more effectively.

One important example of the use of such a mechanism would be in defining a convenient form of variable-length string type, meeting a common need of many users. Following is an example of how such a type might be defined and used:

```
Type VAR_STRING ( LENGTH: NATURAL := 0 ) is
        record
                        VALUE: STRING( 1 .. LENGTH );
        end record;

subtype TEXT_STRING( LENGTH => 1 .. 80 );

subtype COMMAND_STRING is TEXT_STRING( 1 .. 10 );

function VS ( S: STRING ) return VAR_STRING;

LINE : TEXT_STRING;

CS : COMMAND_STRING := VS( "list" );

T1 := VS( "This is a string of text" );

T1 := CS;  -- assignment of a short string to a long string
```

The advantage of this approach over the conventional approach of using a discriminant to express the maximum size of a given object is the compatibility of objects with different subtypes (and hence possibly different maximum sizes). This also can provide significant space savings by not requiring the definition of a single maximum size subtype that must be used for objects of widely varying sizes.

Partially constrained subtypes would also be useful for expressing refinements for variant record types. For example, a compiler might use a variant record for describing symbol table entries and include different variants for different subclasses of symbols:

```
type SYMBOL_CLASS is

        ( NO_SYMBOL, VARIABLE_OBJECT, CONSTANT_OBJECT,
          INTEGER_TYPE, ARRAY_TYPE, RECORD_TYPE, PROCEDURE_UNIT,
          FUNCTION_UNIT, TASK_UNIT, ... );

subtype OBJECT_CLASS is SYMBOL_CLASS
        range VARIABLE_OBJECT .. CONSTANT_OBJECT;

subtype TYPE_CLASS is SYMBOL_CLASS
        range INTEGER_TYPE .. RECORD_TYPE;

subtype UNIT_CLASS is SYMBOL_CLASS
        range PROCEDURE_UNIT .. TASK_UNIT;

type SYMBOL ( CLASS : SYMBOL_CLASS := NO_SYMBOL ) is
        record
          ...
          case CLASS is
                        when OBJECT_CLASS => ...
                        when TYPE_CLASS => ...
                        when UNIT_CLASS => ...
                        when others => ...
          end case;
        end record;

subtype OBJECT_SYMBOL is SYMBOL( OBJECT_CLASS );

subtype UNIT_SYMBOL is SYMBOL( UNIT_CLASS );
```

Objects of subtype UNIT_SYMBOL would be assignable to objects of type SYMBOL, but not to objects of type OBJECT_SYMBOL (the latter assignment would incur an exception).

CURRENT WORKAROUNDS:

Users are forced to use fully unconstrained objects that waste space and don't express intended logical refinements (and allow incorrect assignments to go unchecked) or else to use constrained objects that may also waste space and restrict flexibility.

POSSIBLE SOLUTIONS:

Extend the syntax and semantics of discriminants constraints to permit partially constrained subtypes. This

feature might only be permitted for types whose discriminants have defaults. The syntax could be something like the following:

```
discriminant_constraint ::=

        (discriminant_association {, discriminant_association})
discriminant_association ::=

        [ discriminant_simple_name ] {|discriminant_simple_name} =>
        discriminant_refinement

discriminant_refinement ::=

        expression | discrete_range
```

Compatibility of discriminant range refinements would be checked as part of the elaboration of discriminant constraints and further refinements could be applied for existing partially constrained subtypes. Subtype checking on assignment to an object with a partially constrained subtype would check compatibility of the source value with the target subtype's discriminant ranges.

The principal difficulty of integrating such a feature into Ada is to determine appropriate semantics for compatibility of discriminated subtypes in parameter passing. While it should clearly be possible to associate an object with a formal parameter of mode out (or in out) whose subtype is fully constrained or fully unconstrained, it is less clear what should happen if the formal parameter's subtype is partially constrained. One approach would be to require exact compatibility of subtypes if the formal subtype is partially constrained. Another issue arises when considering compatibility checks for assignments to out parameters when the formal subtype is unconstrained. In order to perform compatibility checks properly when the actual parameter is partially constrained it is necessary for discriminant subtype bounds information to be passed to the subprogram. This clearly requires an extension to current implementation approaches for discriminated parameters, but is solvable without introducing undue distributed overhead for programs not using this feature by careful integration with the existing support already required for the CONSTRAINED attribute.

NESTED VARIANTS

DATE: June 7, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10 - Bus 5
Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 3.7.2

PROBLEM:

Record types with variants work well as long as the variant subcomponents do not contain variants themselves.

```
type T_KIND is          (C_NUMERIC, C_STRING);

type T_NUMERIC is digits 6;

type T_EXPRESSION (KIND : T_KIND) is
        record
                        REFERENCE_COUNT : INTEGER;
                        case KIND is
                                when C_NUMERIC=> NUMERIC_VALUE : T_NUMERIC;
                                when C_STRING => STRING_VALUE : STRING;
        end case;
end record;
```

Is all very nice and is much better than in Pascal or C. But what happens if we start moving towards a more sophisticated implementation than the one given above?

```
type T_NUMERIC_1_KIND is (C_INTEGER, C_REAL);
type T_NUMERIC_1 (KIND : T_NUMERIC_1_KIND) is
        record
                        case KIND is
                                when C_INTEGER => 1_VALUE : INTEGER;
                                when C_REAL  => R_VALUE : LONG_FLOAT;
                        end case;
        end record;
type T_NUMERIC_2_KIND is (C_ZERO_ERROR, C_BOUNDED_ERROR);

type T_NUMERIC_2 (KIND : T_NUMERIC_2_KIND) is
        record
                        VALUE : T_NUMERIC_1;
                        case KIND is
```

```
                        when C_ZERO_ERROR => null;
                    when C_BOUNDED_ERROR => UPPER, LOWER : LONG_FLOAT;
        end case;
end record;

type T_KIND is (C_NUMERIC, C_STRING);
type T_EXPRESSION (KIND : T_KIND) is
        record
                    REFERENCE_COUNT : INTEGER;
                    case KIND is
                        when C_NUMERIC => NUMERIC_VALUE : T_NUMERIC_2;
                        when C_STRING => STRING_VALUE : STRING;
                    end case;
        end record;
```

A little higher level expression (support for a bounded error) and a little more efficient (integer operations can be executed as such). Before considering the problems with this illegal Ada code, one should understand that this is not exotic; one can go much further in this "classification" of a given type, in this case one might add units of measurement and/or support for SHORT_ and LONG_ Ada predefined integer and floating point types, support for fixed point types, support for (software emulated) "infinite range" types, complex numbers etc..., and this just for a single type. In fact, this grouping of related types into a single type, is the cornerstone of object oriented programming in the original meaning (not Booch but Simula-67, SmallTalk-80, Eiffel, C++, Objective-C, ...), so it is worth looking at the problem.

The problem above is that record components need to be constrained if the type has discriminants without default values, so there are three options : either we constrain the values, we add default values, or we allocate each component dynamically.

The problem with dynamic memory allocation of each component is time efficiency : dynamic memory management is not cheap in terms of CPU cost, garbage collection is even more expensive, accessing virtual memory implies paging problems when huge amounts of data are manipulated, and finally the references themselves cost memory. All factors increase at least linearly with the number of objects. Hence, in general, using a dynamic object for each variant component means that each object of a class that is n deep in a specialization tree with be composed of a n separately allocated objects.

The problem with adding default values is purely a memory efficiency problem : when adding default values and using unconstrained record components, one will obtain mutant components that take as much memory as their largest variant, in the example above the 2-byte-integer zero-error numeric (which could be represented byte plus the discriminants) will need 24 bytes (3 * LONG_FLOAT) plus the discriminants, which is about 6 times more.

The problem with constraining the components presents various problems for the programmer. Let's use this solution on the example presented above :

```
type T_NUMERIC_1_KIND is (C_INTEGER, C_REAL);
type T_NUMERIC_1 (KIND : T_NUMERIC_1_KIND) is
        record
                    case KIND IA
                        when C_INTEGER => 1_VALUE : INTEGER;
                        when C_REAL => R_VALUE : LONG_FLOAT;
                    end case;
```

```
        end record;

type T_NUMERIC_2_KIND is (C_ZERO_ERROR, C_BOUNDED_ERROR);
type T_NUMERIC_2 (KIND_2 : T_NUMERIC_2_KIND; KIND 1 :
T_NUMERIC_1_KIND) is
        record
                        VALUE : T_NUMERIC_1 (KIND_1);
                        case KIND is
                                when C_ZERO_ERROR => null
                        when C_BOUNDED_ERROR => UPPER, LOWER : LONG_FLOAT;
                        end case;
        end record;

type T_KIND is ( C_NUMERIC, C_STRING);
type T_EXPRESSION ( KIND : T_KIND;
                                        KIND_2 : T_NUMERIC_2_KIND;
                                        KIND_1 : T_NUMERIC_1_KIND) is
        record
                        REFERENCE_COUNT : INTEGER;
                        case KIND is
                                when C_NUMERIC => NUMERIC_VALUE : T_NUMERIC_2
                                                (KIND_2, KIND_1);
                        when C_STRING => STRING VALUE : STRING;
                        end case;
        end record;
```

One problem is the amount of discriminants that one gathers at the "root" type : there are as many discriminants as there are non-leaves in the specialization tree.

A resulting problem is the meaningless" programing that one needs to initialize a value of the type T_EXPRESSION, e.g.:

```
V : T_EXPRESSION (KIND => C_STRING,
                        KIND_1 => C REAL, -Meaningless but required
                        KIND_2 => BOUNDED_ERROR, -Meaning but required
                                REFERENCE_COUNT => 0,
                                STRING_VALUE => "not so nice");
```

At last, one has the problem of non-modularity. Information about implementations of the numeric expressions should not influence string expressions. The solution presented implies such dependencies.

To summarize the problems involved : one wants efficient access to the components of an "object oriented" type, one wants to represent such type by a single dynamic object, one wants to use unconstrained subcomponent types while still being able to constrain them later.

IMPORTANT: IMPORTANT

CURRENT WORKAROUNDS:

Ignore the problems mentioned above...

POSSIBLE SOLUTIONS:

A "simple" solution might be to have a constraining new operation, which would allow us to constrain all subcomponents of a type, even if they were not constrained in the type's declaration:

allocator:= new subtype_indication | new [constrained] qualified_expression |

The advantages are clear:

The "dynamic" type of an object is fixed at creation time and cannot be modified subsequently (runtime check on 'CONSTRAINED same as Ada-83).

The dynamic object can be represented by the minimal required memory, as if each "dynamic" type would have been a compile-time record type.

The creation operation can now be implemented in modular fashion : one could create an expression by first giving a local mutant the value of the object to be created (an operation which can be distributed over different packages), and subsequently use the mutant in the qualified expression.

Each variant component of the dynamic object could possibly be accessed by first reading the combined value of all discriminants involved, indexing a component location table with this combined value, and adding the result to the object's address. A alternative implementation could use type descriptors. Anyway, there is clearly no implementation or efficiency problem with this approach since a "good" compiler already uses these techniques for records with consecutive subcomponents which constraints depend on the discriminants.

There is no conflict with the current programs which could not have used the new feature, so the only compatibility problem is the new reserved word. Either one accepts the fact that identifiers named "constrained" need to be substituted (not a dramatically complex operation), or one takes an existing keyword (constant, limited, is, ...)

The solution is compatible with Ada's dynamic mutability approach (and in fact only possible due to this approach), and forms a direct extension of the principle; according to the Alsys Rationale, one preferred dynamic mutability to avoid different treatment of objects of mutable and immutable types (in the former case an extra component, here an extra discriminant), and to avoid allocation of unnecessary memory (in the former case because of unconstrained dynamic objects in the case of incompletely (shallow) constrained dynamic objects). A typical argument used at that time was that discriminants of dynamic objects are unlikely to change after allocation, an argument that equally holds here.

When one inspects the domain of object oriented programming a little further, one finds out that in any event, the modularity problem is not solved completely by any modification of the variant records issue. In particular, any object oriented "message" needs to be translated into a subprogram that selects the proper implementation of a "redefineable" method by a case statement that transfers control to the proper subclass package. Hence, adding method implementation redefinitions will, without further adaptation of the language to object orientedness, always affect the class that specifies the method. From that point of view, the modularity issue is something that should be left to the domain of ASPE's and incremental compilers, without affecting the language. Hence, only focusing on the possibility of porting object oriented software to Ada, another alternative could only solve the discriminant flattening problem by introducing variant parts in discriminant_parts. Using the approach, our example could look like :

```
type T_KIND is (C_NUMERIC, C_STRING);
```

```
type T_EXPRESSION (KIND : T_KIND:
        case KIND is
                        when C_STRING => null
        when C_NUMERIC => KIND_2 : T_NUMERIC_2_KIND;
                        KIND_1 : T_NUMERIC_1_KIND;
        end case;) is
record
        REFERENCE_COUNT : INTEGER;
        case KIND is
                        when C_NUMERIC => STRING_VALUE " T_NUMERIC_2 (KIND_2,
                                                KIND_1);
                        when C_STRING => STRING_VALUE : STRING;
        end case;
end record;

E := new T_EXPRESSION( C_STRING);
E := new T_EXPRESSION( C_NUMERIC, KIND_1 =>C_REAL, KIND_2 => C_BOUNDED_ERROR);
```

This approach would introduce less new concepts, and is consistent with the approach taken for variants in general : types remain entirely constrained or not constrained at all, and all components (this time including discriminants) that depend on discriminant values exist if and only if the discriminants have the value they depend on.

A slightly modified version of this solution is also possible : instead of imposing the specification of all dependent discriminants that are used to constrain dependent subcomponents, one could define the as existing, implicit discriminants.

One could introduce the concept of explicit and implicit discriminants : explicit discriminants are those found in the type declarations, implicit discrimi ants are discriminants of all subcomponents that were not constrained and should constrained according to the Ada83 standard. When declaring an object of a type, one should specify all explicit and implicit discriminants.

```
type T_KIND is (C_NUMERIC, C_STRING);
type t_EXPRESSION (KIND : T_KIND) is
        record
                        REFERENCE_COUNT : INTEGER;
                        case KIND is
                                when C_NUMERIC => NUMERIC_VALUE : T_NUMERIC_2;
                                                --Not allowed in Ada83
                                when C_STRING => STRING_VALUE : STRING;
                        end case;
        end record;

E := new T_EXPRESSION( C_STRING;
E := new T_EXPRESSION( C_NUMERIC, KIND_1 =>C_REAL,KIND_2 =>C_BOUNDED_ERROR);
                        --- Required values for implicit discriminants
```

At the declaration this comes to exactly the same result, the only difference is that one does not explicitly repeat the existence of these nested discriminants : easier to write and maintain, more difficult to read. This lat drawback is probably a decisive to prefer the solution mentioned earlier.

Possibly even better solutions can be found, but unless clear drawbacks of the suggested solution can be demonstrated, no solution will always be worse than a naive and/or inelegant solution.

A last indication that the variant records issue is not entirely "clean is the fact that I couldn't find any literature that seriously deals with this topic. Only the Rationaly mentions that problem of variant record, but, with all due respect for the already elaborate discussion found there it does not consider the problem of nested variants at all.

ALLOW ARRAY TYPE DEFINITIONS WITHIN RECORDS

DATE: April 24, 1989

NAME: Bjorn Kallberg

ADDRESS: Ericsson Radar Electronics
S-164 84 Stockholm
Sweden

TELEPHONE: +46 8 757 35 08
+46 8 752 81 72
E-mail: ada_ubk@kierre.ericsson.se

ANSI/MIL-STD-1815A REFERENCE: 3.7 (2)

PROBLEM:

With the current record type definition, it is impossible to define a general type with variable size in two or more dimensions. This was legal in the 1980 definition. It was probably inadvertently removed, in a general attempt to decrease the use of anonymous array types.

One of the greatest advantages of Ada is the possibility to build composite, general, parametrized types. Presently, this facility is limited to one dimension. The building of higher levels of abstraction, using simpler types, is often impossible.

Example: You want to define a page of text, naturally consisting of a number of lines. For lines you want to use the line definition from a package such as the Text_handler package in LRM 7.6. However,

```
type Page_t(lines_on_page, Chars_on_line : integer) is record
   Lines : array(1..Lines_on_page) of
                        Text_handler.text(Maximum_length => Chars_on_line);
end record;
```

is illegal.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Hard code the size of the lines. Hardly portable or elegant.

Use access types. Gives different semantics and makes garbage collection necessary.

Default on discriminant. Compiler either allocates maximum size of record, thus inefficient use of memory, or allocation on heap, with same problem as above.

POSSIBLE SOLUTIONS:

Return to the 1980 definition of record type, i.e., add the last line in the definition below.

component_ declaration ::=
        identifier_list : component_subtype_definition [:=expression]
!   identifier_list : array_type_definition [:=expression]

MULTIPLE NON-NESTED VARIANT PARTS FOR RECORD TYPES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3.7(2)

PROBLEM:

A discriminated record type should be allowed to have multiple non-nested variant parts. For example, the following should be legal:

```
type Foo_Kind is (A,B,C,D);
type Foo (Kind : Foo_Kind) is
        record
                        case kind is
                                when A | B =>
                                        F1 : Integer;
                                when others =>
                                        null;
                        end case;

                        case kind is
                                when A | C =>
                                        F2 : Float;
                                when others =>
                                        null;
                        end case;

                        case kind is
                                when B | C =>
                                        F3 : Duration;
                                when others =>
                                        null;
                        end case;
        end record;
```

This would be much more readable and convenient (and probably more efficient) than the alternative that is currently available:

```
type Foo_Kind is (A,B,C,D);
type Variant_Part_1 (Kind : Foo_Kind) is
```

```
        record
                        case kind is
                                when A | B =>
                                        F1 : Integer;
                                when others =>
                                        null;
                                end case;

                        end record;

type Variant_Part_2 (Kind : Foo_Kind) is
        record
                        case kind is
                                when A | C =>
                                        F2 : Float;
                                when others =>
                                        null;
                        end case;
        end record;

type Variant_Part_3 (Kind : Foo_Kind) is
        record
                        case kind is
                                when B | C =>
                                        F3 : Duration;
                                when others =>
                                        null;
                        end case;
        end record;

type Foo (Kind : Foo_Kind) is
        record
                        Vp_1 : Variant_Part_1 (Kind => Kind);
                        Vp_2 : Variant_Part_2 (Kind => Kind);
                        Vp_3 : Variant_Part_3 (Kind => Kind);
        end record;
```

This sort of problem can arise fairly often in practice, whenever one defines a data structure that is built up of different kinds of nodes along with various attributes, each of which is defined only on nodes of a particular set of node_kinds.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Introduce intermediate discriminated types and use propagated discriminant constraints, as in the previous example.

POSSIBLE SOLUTIONS:

ENDING RECORD DECLARATIONS WITH TYPE NAME ITSELF

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.7(2)

PROBLEM:

Some record declarations can get quite long, and it is hard to remember at the end what the record was called at the start.

IMPORTANCE: IMPORTANT

This revision improves readability of the code.

CURRENT WORKAROUNDS:

Some programmers comment the bottom of long record declarations with the name of the record, as follows:

```
type Foo is
        record
                ...
        end record; --Foo
```

POSSIBLE SOLUTIONS:

It should be possible to end record declarations with the name of the record, instead of with the reserved word RECORD:

```
type Foo is
        record
                ...
        end Foo;
```

This should be optional, with the reserved word RECORD still be acceptable.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will re-compile successfully and will behave identically during execution except for possible small changes in execution speed.

THE IDENTIFIERS OF ALL COMPONENTS OF A RECORD TYPE MUST BE DISTINCT

DATE: July 12, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10 - Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 3.7(3)

PROBLEM:

The identifiers of all components of a record type must be distinct. This restriction is a very tricky one, it is probably accepted because it was considered indispensable for type checking reasons. But what happens in practical situations as a result of this rule? If the rule would be replaced by a rule that focusses on type checking only, one could write a (perfectly unambiguous) program like :

```
procedure TEST is
        type T_KIND is (A,B,C);
        type T(KIND : T_KIND :=A) is
                        record
                                U : INTEGER;
                                case KIND is
                                        when A => X,Y : INTEGER;
                                        when B => X,Z : INTEGER;
                                        when C => X,Z : INTEGER;
                                end case;
                        end record;
V : T;
function TWICE_X (V : T) return INTEGER is
begin
        return V.X * 2;
end;
procedure SET_X (V : in out T; VALUE : INTEGER) is
begin
        V.X := VALUE;
end;
```

Remark that this example conforms to the Ada design because the type of V.X is statically known.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Shown above. With the rule 3.70(3), the program shown above is illegal. The workaround is simple, mechanical and cumbersome : one needs to combine the name of the discriminant value and the component and use a case for each access:

```
procedure TEST is
        type T_KIND is (A,B,C);
        type T (KIND : T_KIND := A) is
                        record
                                U : INTEGER;
                                case KIND is
                                        when A => A_X,A_Y : INTEGER;
                                        when B => B_X,B_Z : INTEGER;
                                        when C => C_Y,C_Z : INTEGER;
                                end case;
                        end record;
V : T;
function TWICE_X (V : T) return INTEGER is
begin
        case V.KIND is
                        when A => return V.A_X * 2;
                        when B => return V.B_X * 2;
                        when C => raise CONSTRAINT_ERROR;
        end case;
end;
procedure SET_X (V : in out T; VALUE : INTEGER) is
begin
        case V.KIND is
                        when A => V.A_X := VALUE;
                        when B => V.B_X := VALUE;
                        when C => raise CONSTRAINT_ERROR;
        end case;
end;
...
```

I believe that the example illustrates that such kind of problems should really be dealt with by a machine and not by a programmer.

POSSIBLE SOLUTIONS:

Remove rule 03.07(03) and replace it with a milder rule:

> Each identifier used in the identifier_list of a component_declaration should be associated with a single type within a single record_type_definition.
>
> The identifiers used in a component_list should be all distinct, and should not be used in subsequent variant_parts.

This rule assures that any selection that selects a component with an identifier that has been used more than one in a record declaration has a type that is statically known, and denotes at most one subcomponent of the prefix.

NON-DISTINCT RECORD COMPONENT IDENTIFIERS

DATE: October 23, 1989

NAME: David Calloway, Randall Rushe

ADDRESS: SAIC (Science Applications International Corporation
5517 Hickory St.
Panama City, FL 32404

TELEPHONE: (904) 784-1799

ANSI/MIL-STD-1815A REFERENCE: 3.7(3)

PROBLEM:

The current ANSI/MIL-STD-1815A requires all record component identifiers to be distinct, even when one identifier would logically apply to more than one, but not all variants in the record. The proposed change would relax this restriction by permitting non-distinct component identifiers to appear in more than one variant in the declaration of the type, as long as any elaboration of the record would not result in multiple components with the same identifier.

An example of this would be the following declaration of a coordinate system record which would not be a valid record type declaration under the current standard, but would be permitted if the proposed change were approved:

```
type Coordinate_System_Type is ( Spherical, Cylindrical, Cartesian);

type Coordinate_Record (Coordinate_System :     Coordinate_System_Type) is
        record
                        case Coordinate_System is
                                when Cartesian =>       X: Float;
                                                        Y: Float;
                                                        Z: Float;

                                when Spherical =>       R: Float;
                                                    Theta : Float;
                                                      Phi: Float;

                                when Cylindrical =>     R: Float;
                                                    Theta : Float;
                                                        Z: Float;
                        end case;
        end record;
```

IMPORTANCE: IMPORTANT

This problem arises with some regularity, and the existing restriction thwarts efforts to have meaningful and intuitive identifiers. One of the best features of the Ada language is its capability to reflect the characteristics of the problem for which a program is written. The proposed change will enhance Ada's

strong self-documenting and strong-typing features.

CURRENT WORKAROUNDS:

There are two current workarounds to the problem:

1. Change one or more of the identifiers to maintain uniqueness. This fully adheres to the current standard, but results in the non-intuitive use of different identifiers to represent logically similar components, or the use of identifiers which do not map closely to the problem space.

Example:

```
type Coordinate_Record ( Coordinate_System :   Coordinate_System_Type) is
        record
                        case Coordinate_System is
                                when Cartesian =>       X : Float;
                                                        Y : Float;
                                                        Z : Float;

                                when Spherical =>       R : Float;
                                                    Theta : Float;
                                                      Phi : Float;

                                when Cylindrical =>  R_Cyl: Float;
                                                 Theta_Cyl : Float;
                                                     Z_Cyl: Float;
        end case;
end record;
```

2. Do not use variant parts. Instead, specify components which will satisfy all cases. This requires that storage for all components be allocated for an object whether applicable or not and, again, results in a data structure that does not map directly to the problem space. This makes the resulting code more difficult to understand and requires users of the record to specify values for identifiers that do not apply to the specific instance of the record they have elaborated.

Example:

```
type Coordinate_Record is
        record
                                X  : Float;
                                Y  : Float;
                                Z  : Float;
                                R  : Float;
                            Theta  : Float;
                              Phi  : Float;
        end record;
```

3. Use a combination of variant and non-variant parts, placing identifiers which occur in only one variant in the variant part and identifiers which would logically appear in more than

one variant in the non-variant part. The disadvantage with this solution is that it again does not map elegantly to the problem space, makes the resulting code more difficult to understand, and prevents the compiler from detecting invalid combinations that could be specified in this format.

Example:

```
type Coordinate_Record ( Coordinate_System :   Coordinate_System_Type) is record

        Z           : Float; --    Should only be used in Cartesian and
                             --    Cylindrical coordinate systems.

        R           : Float; --    Should only be used in Spherical and
                             --    Cylindrical coordinate systems.

        Theta       : Float; --    Should only be used in Spherical and
                             --    Cylindrical coordinate systems.

        case Coordinate_System is

                when Cartesian =>     X :  Float;
                                      Y :  Float;

                when Spherical => Phi :  Float;
                when Cylindrical  => null;
        end case;

end record;
```

POSSIBLE SOLUTIONS:

Preferred Solution:

Revise the standard to permit non-distinct component identifiers, so long as any elaboration of the record does not result in multiple components with the same identifier. A change in wording of the standard might be:

"The identifiers of all components of a record type must be distinct [for any elaboration of the record type definition]." (LRM 3.7)

An additional reasonable restriction is that there be no potential for an elaboration to result in non-distinct identifiers. This permits full compile-time checking. Sample wording might be:

"The identifiers of all components of a record type must be distinct [for any POSSIBLE elaboration of the record type definition]." (LRM 3.7)

and

"[Non-distinct component identifiers may not appear in the same variant of a variant part and may not appear outside a variant part. Nor may the same component identifier appear both inside and outside a variant part.]" (LRM 3.7.3)

"[Non-distance component identifiers may only appear in different variants of a variant part.]"

A further restriction that all components having the same identifier must also have the same type would be acceptable in most cases, but is probably not necessary for implementation.

Examples:

An example of a proposed legal record type definition is the one presented in the problem statement. Another would be:

```
type DISCRIMINANT_TYPE is ( SET_A, SET_B, UNION_A_B);
type GOOD_RECORD (DISCRIMINANT : DISCRIMINANT_TYPE ) is
        record
                        case DISCRIMINANT is
                          when SET_A        => COMPONENT_A : COMPONENT_TYPE;
                          when SET_B        => COMPONENT_B : COMPONENT_TYPE;
                          when UNION_A_B    => COMPONENT_A : COMPONENT_TYPE;
                                               COMPONENT_B : COMPONENT_TYPE;
                        end case;
        end record;
```

The following would be an illegal record type definition (even though a record elaborated with SET_A would not have non-distinct components).

```
type BAD_RECORD ( DISCRIMINANT : DISCRIMINANT_TYPE ) is
        record
                        COMPONENT_B : COMPONENT_TYPE;
                        case DISCRIMINANT is
                          when SET_A        => COMPONENT_A : COMPONENT_TYPE;
                          when others       => COMPONENT_B : COMPONENT_TYPE;
                        end case
        end record;
```

Alternative solution:

If multiple variant parts were permitted there would be several possible ways of declaring the problematic record structures, without requiring non-distinct identifiers. However, these are less straightforward than for the preferred solution, and may require a more complex implementation. For example:

```
type GOOD_RECORD_2 ( DISCRIMINANT : DISCRIMINANT_TYPE ) is
  record
        case DISCRIMINANT is
                        when   SET_A   |   UNION_A_B   =>   COMPONENT_A   :
                        COMPONENT_TYPE;
                        when SET_B              => null;
        end case;

        case DISCRIMINANT is
                        when SET_A              => null;
                        when   SET_B   |   UNION_A_B   =>   COMPONENT_B   :
```

```
                                    COMPONENT_TYPE;
            end case;
        end record;
```

Impact:

The proposed solutions would be upward compatible with all existing Ada application code, and should have minimal impact on compiler implementations.

USE OF ACCESS VARIABLES TO REFERENCE OBJECTS DECLARED BY OBJECT DECLARATIONS

DATE: June 9,1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 3.8

PROBLEM:

In the Notes for section 3.8 is the statement 'An access value can only designate an object created by an allocator; in particular, it cannot designate an object declared by an object declaration.' This statement, while reflecting the intentions of the designers of the language, has proven to be inconsistent with the usage of the language. In order to set up linked lists in which all elements are static and to point to the next object in the list, one can either use access objects or objects of type ADDRESS. Use of ADDRESS has been objected to on grounds of lack of type-checking. After initialization, the access object is used generally in accordance with the intent of the language designers.

IMPORTANCE: IMPORTANT

The semantics of the allocator and ADDRESS capabilities provided by the language pose unnecessary obstacles to applications requiring the use of static objects in linked lists. At the same time, these obstacles decrease the amount of compile time checking that can be done.

CURRENT WORKAROUNDS:

Use unchecked conversion between ADDRESS and access types to initialize access objects to reference declared objects. However, there is not required checking that the address of the source operand to the unchecked conversion function actually references an object of the target access type.

POSSIBLE SOLUTIONS:

(1) Delete the statement in the Notes section referenced above. It is misleading at best, and is more accurately viewed as being inaccurate.

(2) Allow explicit conversion between objects of type ADDRESS and access objects as long as the target of the access type is the same as the object whose ADDRESS is being converted.

See MOWDAY-002 for a related submittal.

KNOWING IF GARBAGE COLLECTION IS BEING PERFORMED

DATE: October 24, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail: madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 3.8

PROBLEM:

Ada 83 allows too much freedom in the memory allocation/ deallocation mechanisms. This affects the portability of programs that use access types.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Avoid access types and task types.

POSSIBLE SOLUTIONS:

The language standard should provide an attribute that allows an application to know if it performs garbage collection on a given collection, or if Unchecked_Deallocation must be used. When garbage collection is not performed, Unchecked_Deallocation should be required to do something more than just setting its parameter to null; that is, to actually deallocate the storage occupied by the designated object. It must be recognized, however, that this would be hard to check in a validation suite, but it is very important in the case of long-running programs. The language standard should also specify that constructs other than allocators are not allowed to generate memory leaks.

GARBAGE COLLECTION IN ADA

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 3.8, 4.8, 13.2, 13.10.1

SUMMARY:

The general solution to the allocation of storage for accessed variables in Ada is garbage collection. Garbage collection is able to accurately determine which portions of storage are still accessible to the program and which are not, so that the inaccessible portions can be returned for reuse by the running program. Because garbage collection is "underneath" the program, it is capable of examining the entire state of the program, and can therefore manipulate these "inaccessible" objects.

During the first decade of Ada's existence, Ada implementors and Ada programmers were encouraged to stay away from garbage collection because of the "horror stories" that they had been told. Garbage collection is "inefficient", can cause a "real-time" program to go temporarily "catatonic", and involves a commitment to a larger run-time system.

We suggest that the time has come to reconsider this recommendation, and to suggest that Ada 9X encourage implementors and programmers to use garbage collection, including for mission-critical real-time systems.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The Ada language provides for very flexible data structures to be built using access types. The storage required for these structures is acquired from the Ada run-time system by the "new" construct, which allocates the space. The space is then occupied until Ada can prove that no harm would come from reusing the storage for something else. Typically this proof comes in the form of a proof that the storage is no longer accessible, the contents of the storage can have no further bearing on the course of the computation.

(There may be a case where an active task becomes inaccessible itself, but which is still capable of changing the values of data visible to others. In this case, the storage should not be reclaimed. This issue is address in [Baker77].)

Proofs of inaccessibility are typically done by a garbage collector. Two standard techniques are in use -- the reference count technique and the mark-sweep technique. The reference count technique keeps count within each object of the number of external references to the object. When this count becomes zero, the object is inaccessible [Deutsch]. However, if there are directed loops within a structure, the reference count will not go to zero, and these loops cannot be easily reclaimed (however, see[Bobrow]). The mark-sweep method traces out all possible access paths, marking all accessible nodes as it goes. Those nodes left

unmarked at the end of the marking process are by definition inaccessible. [Cohen] is a good survey of garbage collection strategies.

Both the reference count and the mark-sweep techniques have been made into "incremental" and "real-time" methods. The reference count technique is already quite incremental in its nature, and its behavior can be adjusted to tailor the amount of work performed on an allocation so that it is proportional to the amount of space requested [Baker78]. (Making reference count truly incremental requires deferring the disassembly of returned objects until their storage is needed.) The mark-sweep technique can also be made incremental [Baker78], although requiring a slight increase in complexity, but its behavior is similar in that its work on allocation is proportional to the amount of space requested.

What is less well known, is that tight bounds can be placed on the amount of time required to allocate a block of size n in an incremental mark-sweep method, where the time bound is of the form Cn, for some (surprisingly small) constant C [Baker78]. A system can be built using this technique that satisfies hard real-time requirements, assuming that there is an upper bound on the sizes of the objects being requested. Furthermore, these time bounds are the best that can be achieved, in the sense that any sufficiently general storage allocation mechanism must take time proportional to the size of the storage being allocated to perform its task. (Even if the allocation of storage could be done faster, most systems could not take advantage of this, because the initialization of that storage requires time proportional to the storage allocated.)

Most of the storage allocation techniques utilized by operating systems or Ada run-time systems cannot claim "real-time" behavior. Any Ada run-time system that does not relocate objects (nearly all of them) must deal with storage fragmentation. Fragmented storage implies a fragmented free-list, and free-lists must be searched to find a portion of storage which can accommodate a request. For example, DEC Ada utilizes a "find-first-fit" search for its Ada [DEC]. A search through a fragmented free storage list takes a certain amount of time, and this time in most systems does not have an upper bound, but is statistically reasonably small.

An Ada run-time system that does not relocate objects (nearly 100% of the Ada implementations) cannot use storage efficiently, and in the worst case can use storage extremely inefficiently. For example, [Robson] shows that an allocation of a block of size $\leq n$ can fail even when memory is only about $1/(1+.5\log n)$ full; if $n=64$, allocation can fail when memory is only 25% occupied, and if $n=1024$, allocation can fail when memory is only 16.7% full. Robson's result is a combinatorial result about the block sizes themselves, and has nothing to do with a garbage collection algorithm; it therefore applies to all Ada systems. In any particular system, a storage allocation system can fail at even small occupancy levels; Robson's result says that no matter how good the allocation/deallocation system is, one can force it to fail at the above-mentioned occupancy levels by a suitably chosen sequence of allocation and deallocation requests. In other words, storage can become very badly fragmented, implying relatively long free-list searches.

An Ada run-time system which does relocate must update all of the pointers to updated objects, and finding these pointers may be tantamount to implementing a general mark-sweep garbage collection algorithm.

Note that a reference count system-- in addition to having problems with directed cycles--does not usually relocate objects, so that it must contend with all of the storage fragmentation problems talked about above, which in turn implies not-very-real-time allocation behavior.

The net result of these observations is that while there are contenders for an Ada garbage collection system, there is only one contender that offers the possibility of hard real-time behavior--the incremental relocating mark-sweep approach [Baker78].

For non-embedded Ada systems such as programming environments, the "generational garbage collectors" have become the state-of-the-art [Lieberman, Unger, Moon, Courts]. They offer superior average performance, as well as superior locality of reference in a paging environment, and therefore are attractive for non-real-time and non-real-memory environments. They can also be efficiently implemented using the stock memory mapping hardware of modern computers [Appel, Shaw]. However, these approaches still cannot offer hard real-time capabilities unless their operation is also incremental in nature. However, commercial systems incorporating garbage collection have reached acceptable real-time performance for systems environments; the Symbolics Lisp Machines operate continuously as file servers, even when they are continuously collecting garbage [Moon].

The net result of the past 10 years of research and implementation experience in garbage collectors is thus:

* Garbage collection is compatible with the requirements of hard real time systems.

* Garbage collection is an accepted implementation technique for several popular languages.

* Generational garbage collection is statistically very efficient, and many commercial systems use it daily.

* Garbage collection is no longer the specter that it appeared when Ada was standardized.

CURRENT WORKAROUNDS:

Workarounds have been discussed at great length in the literature. Most workarounds eventually cost as much effort as simply biting the bullet and offering garbage collection.

NON-SUPPORT IMPACT:

Less productive systems designers who must constantly scratch their heads to figure out how to avoid garbage collection. More buggy systems that opt for "unchecked deallocation". Heavier maintenance costs when someone must find the one place in a 10,000,000+ line program where someone screwed up and deallocated the wrong thing.

POSSIBLE SOLUTIONS:

Discussed above.

DIFFICULTIES TO BE CONSIDERED:

Distributed systems are an interesting problem, but garbage collection in those environments has been studied [Halstead].

REFERENCES:

Appel, Andrew W., Ellis, John R., and Li, Kai. "Real-time concurrent garbage collection on stock multiprocessors". ACM Prog. Lang. Des. and Impl., June 1988, p.11-20.

Baker, Henry G., Jr. "The Incremental Garbage collection of Processes". ACM Symp. on AI and Prog. Langs., also SIGPLAN Notices I2, 8 (Aug 1977), pp.55-59.

Baker, Henry G., Jr. "List processing in Real Time on a Serial Computer". CACM 21, 4 (April 1978), pp.280-294.

Bobrow, Daniel G. "Managing reentrant structures using reference counts". ACM TOPLASS 2,3 (July 1980), pp.269-273.

Cohen, Jacques. "Garbage collection on linked data structures". Computing Surveys I3, 3 (Sept. 1981), pp.341-367.

Courts, Robert. "Improving Locality of Reference in a Garbage-Collecting memory Management System". CACM 31, 9 (Sept. 1988), pp. 1128-1138.

DEC. VAX Ada and VAXELN Ada Technical Summary. Digital Equipment Corporation order number: AA-GS98A-TE, October, 1985.

Deutsch, L. Peter, and Bobrow, Daniel G. "An efficient, incremental garbage collector". CACM 19, 9 (Sept.1976), pp.522-526.

Halstead, R. "Implementation of Multilisp: Lisp on a multiprocessor". ACM Symp. on Lisp and Func. Prog., Austin, TX, Aug. 1984, pp.9-17.

Liebermann, Henry and Hewitt, Carl. "A real-time garbage collector based on the lifetimes of objects". CACM 26.6 (June 1983), pp.419-429.

Moon, David. "Garbage collection in a Large Lisp system". ACM SIGPLAN Symp. on Lisp and Func. Prog., 1984, pp.235-246.

Robson, J.M. "Bounds for Some functions Concerning Dynamic Storage Allocation". JACM 21,3 (July 1974), pp.491-499.

Shaw, Robert A. "Improving garbage collector performance in Virtual Memory". Stanford University computer Systems Lab. CSL-TR-87-323, March 1987.

Ungar, David. "Generation scavenging: A non-disruptive, high performance storage reclamation algorithm". ACM Sigsoft/Sigplan Symp. on Prac. Soft. Devel. Envs., 1984, pp. 157-167.

HEAP MANAGEMENT IMPROVEMENTS

DATE: October 9, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 3.8, 4.8 (7...12), 7.4, 13.10.1

PROBLEM:

Many current compilers do not provide automatic heap reclamation. This shortcoming encourages the use of UNCHECKED_DEALLOCATION and/or discourages the use of access types. The language does not support all of the constructs needed to perform the necessary checks.

IMPORTANCE: ESSENTIAL

In the embedded systems domain, correct reclamation of the heap is absolutely necessary.

CURRENT WORKAROUNDS:

Extensive testing is used whenever UNCHECKED_DEALLOCATION is present.

POSSIBLE SOLUTIONS:

Allow specification of procedures that are to be called when an object ceases to be visible. Such procedures must accept a single parameter of the appropriate type with in out mode and perform the needed operations. Note that this is also a method of implementing automatic file closing.

Allow a higher degree of detail in pragmas instructing the compiler on heap management. For instance, a simply linked list member object could be managed automatically by the compiler via access count recording.

INCOMPLETE TYPE DECLARATIONS

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.8.1

PROBLEM:

Incomplete type declaration can cause maintenance problems when the user expects an existing unit to be mature and a candidate for reuse. The reused program should be updatable with good confidence of not inserting errors. However; currently, the compilation system does not have to provide warnings when an unreferenced incomplete type declaration is in error. Instead of allowing incompletely defined declarations, the language could merely allow forward references in the declaration syntax/ semantics. Such an approach would cut out the verbosity of an Ada program, remove opportunities to leave incomplete declarations in a program that aren't flagged by the compilation system, etc. Most compilation systems are multipass and can resolve forward references within the same compilation unit.

IMPORTANCE: IMPORTANT

To ensure highly reliable units for further reuse.

CURRENT WORKAROUNDS:

Try to restrain programmers from leaving incomplete type declarations that are not used through the project programming standards. Such capability should not be allowed within language constructs without appropriate error management. It should be avoided in producing programs that must attain high levels of reliability, e.g., embedded processors. New tools may be needed that duplicate much of the parser for recognizing such declaration fragments.

POSSIBLE SOLUTIONS:

1. Rewrite the syntax/semantics to support forward references.

2. Provide for compilation information messages when incompletely specified items remain in the code that are not used.

ACCESS VALUES THAT DESIGNATE CONSTANT OBJECTS

DATE: August 10, 1989

NAME: Donald R. Clarson

ADDRESS: Teledyne Brown Engineering
151 Industrial Way East
Eatontown, NJ 07724

TELEPHONE: (201) 389-6756

ANSI/MIL-STD-1815A REFERENCE: 3.8.1

PROBLEM:

All objects designated by access values are treated as variables.

Access types provide both dynamic allocation (and storage management) of objects during program execution and a means to create data structures with components which refer to other data structures.

Application programs may require access values which designate constant objects so that data structures may be created which refer to these constant values and to allow these access values to be passed as parameters. It may be necessary to preallocate these constant objects and store the collection in a Read-Only-Memory device in order to minimize the time required for elaboration of library packages before system operation begins.

The Ada program will continue to treat these objects as variables and will allow subsequent assignment to these designated objects. Since the Read-Only-Memory device will not store the value assigned, erroneous conclusions could be drawn from an analysis of the program execution.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None. Some environments allow named constants to be stored in ROM with access values created by UNCHECKED_CONVERSION to type SYSTEM.ADDRESS. These access values may then be used in the construction of data structures which refer to these constant values. Ada will treat the designated object as a variable and will not enforce the constant nature of these objects.

POSSIBLE SOLUTIONS:

Add a construct to the language which will allow an access type which may only designate constant objects. A possible syntax for these access type definitions is:

```
type Identifier is access [constant] Subtype_indication;
```

The semantics require that every allocator for this type must provide a qualified expression for the value of the designated object. The language would preclude any subsequent assignment to an object designated

by an access value of this type.

A representation clause which provides the required address in storage for a collection would facilitate the preallocation of these objects in a Read-Only-Memory device. The following paragraph (to follow ARM 13.5(6)) would allow this:

(d) Name of an access type: The address that is required for the collection needed to contain all objects designated by the access type.

INCOMPLETE TYPES CAN'T BE USED ACROSS PACKAGES

DATE: June 27, 1989

NAME: Stef Van Vlierberge

ADDRESS: S.A. OFFIS N.V.
Wetenschaptr. 10 - Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 3.8.1

PROBLEM:

This is one of the most fundamental problems with Ada's modularity support. In short one can say that it is impossible to implement full and efficient support for a binary 1-to-N or N-to-M between types T1 and T2 unless you either imply that T1 and T2 are declared in one declarative region or that one fiddles the language (using type SYSTEM_ADDRESS). The most amazing aspect of this problem is the number of Ada experts who ignore the problem, or propose solutions that just can't work.

A more detailed description is presented by following example:

```
package PERSONS_HAVE_CARS is
        type T_PERSON is limited private;
        type T_CAR is limited private;

        -- Support for Persons
        procedure CREATE_PERSON (P : in out T_PERSON; NAME...);
        function NAME (P_PERSON...

        -- Support for Cars
        procedure CREATE_CAR (C : in out T_CAR; COLOR...);
        procedure COLOR (C_CAR...

        -- Support for the 1 to many relationship between PERSONS and CARS

                procedure TRANSFER_CAR_TO_PERSON (C: T_CAR; P : T_PERSON);
        --Makes P the new owner of C

        function HAS_OWNER (C:T_CAR) return BOOLEAN;
        procedure GET_OWNER (C:T_CAR; P: in out T_PERSON);

        generic
                with procedure FOR_EACH (C: T_CAR);
        procedure FOR_ALL_CARS_OF (P:T_PERSON);
        -- Performs FOR_EACH for each car of P.
```

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 4. NAMES AND EXPRESSIONS

CREATING STUBS

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.9 #9

PROBLEM:

In realtime or large scale developments with strict configuration management (CM) requirements, the process of creating an artificial stub interferes with the development process. It seems unnecessary to have to create a stub with all the capability of the language for separate compilation coupled with the specifications. Stubs are part of some development processes, but seldom useful in weapon systems developments. For every update to a program library unit, proper CM rules must be followed. Therefore, enlarging a stub takes more effort than to create it, install it, and revise it in the program library.

IMPORTANCE: IMPORTANT

The linker can provide dummy calls and returns so that stub does not have to be created and early development and integration can begin.

CURRENT WORKAROUNDS:

Avoid the stub generation or provide additional overhead in the configuration management for updating a stub to a full unit.

POSSIBLE SOLUTIONS:

Allow the compiler to have a parameter for compilation to "assume" a stub for the specification without the user having to create the extra body with only a NULL statement inside.

For additional references to Section 3. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ARRAY PROCESSING

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 4

PROBLEM:

Ada has no facilities for array processing required in numerical computing, circuit design, etc.

IMPORTANCE: IMPORTANT

Without appropriate features in the language, there will be a tendency to write such applications in languages with better array handling, e.g., APL, PL/I or vector FORTRAN.

CURRENT WORKAROUNDS:

Packages using loops. This makes it difficult for the compiler to optimize code, and requires a large number of explicit conversions in the code of the client procedure.

POSSIBLE SOLUTIONS:

Provide functions and operators for the more important array operations, e.g., inner product, outer product, relations (element by element, not just lexicographic), reductions and selection. APL is a good model here, except that the total number of operators should be kept more manageable.

The facilities added should be generalized, as in APL, e.g., "inner product" includes an "or reduction" of the "and" of two boolean matrices.

SYNTAX FOR INDEXED COMPONENTS

DATE: October 31, 1989

NAME: Joe Cross

ADDRESS: Unisys Computer Systems Division
MS U2F13
PO Box 64525
St. Paul MN 55164-0525

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: 4.1

PROBLEM:

SUMMARY:

Readability of programs would be improved by permitting an implementation to accept square brackets as replacements for parentheses in indexing and slicing operations.

IMPORTANCE:

SPECIFIC REQUIREMENT/SOLUTION CRITERIA:

Understanding an Ada program requires the reader to distinguish array indexing and slicing operation from function calls and type conversions. This can currently require the reader to understand a large amount of non-local context.

CURRENT WORKAROUNDS:

JUSTIFICATION/EXAMPLES/WORKAROUNDS:

The expression

```
A(B)
```

could represent an indexed array, a slice, a function call, or type conversion.

NON-SUPPORT IMPACT:

Initial understanding of Ada programs is harder than it has to be.

POSSIBLE SOLUTIONS:

Permit Ada implementation to accept brackets as alternative to parentheses in array indexing and slicing operations.

DIFFICULTIES TO BE CONSIDERED: NONE

REFERENCES/SUPPORTING MATERIAL: NONE

THE SYNTAX FOR SLICES IS TOO RESTRICTED

DATE: August 31, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 4.1.2(3)

PROBLEM:

Slices are allowed only for single-dimensional arrays. Note that other languages allow non-contiguous slicing on multiple indices concurrently, either for the entire range, e.g., PL/I, or for a specified subrange, e.g., ALGOL 68. The ability to do this in Ada would be important, but, at a minimum, slicing on the last index is essential.

IMPORTANCE: ESSENTIAL

Regularity is compromised without this, and the design goals of Steelman and of 1.3(3) are violated:

Concern for the human programmer was also stressed during the design. ... underlying concepts integrated in a consistent and systematic way. ... language constructs that correspond intuitively to what the users will normally expect.

CURRENT WORKAROUNDS:

Arrays of arrays can be used, but Ada does not allow array types with unconstrained components. Record types can be used, but this can cause other problems, due to the restrictions on the use of discriminants. In practice, this restriction usually forces the use of a for ... loop block, leading to less efficient code that is harder to read and to maintain.

POSSIBLE SOLUTIONS:

1. Change 4.1.2(2) to read

slice ::= prefix({expression,}discrete_range)

Change 4.1.2(3) to read

The prefix of a slice must be appropriate for an array with the specified number of subscripts....

2. Change 4.1.2(1) to read

A slice denotes a subarray of an array, indexed by a cartesian product of sequences of indices for one or more subscripts....

Change 4.1.2(2) to read

slice ::=

prefix(discrete_range|expression{,discrete_range|
expression})

change 4.1.3(7) to read

The selector must be a simple name denoting a component or slice of a record object or value. The prefix must be appropriate for an object of this type.

Note that solution 1 is essentially free; solution 2, although more useful, would require considerable more work in order to identify ramifications and consistently update the LRM, as well as requiring more of the implementation.

SLICES OF MULTIDIMENSIONAL ARRAYS

DATE: October 23, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail : madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 4.1.2

PROBLEM:

Slices are available only for one-dimensional arrays. For uniformity and efficiency reasons (especially for vector processors), it would be very useful to have a similar facility for arrays with more than one dimension.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use 'for' loops for copying subarrays, which reduces readability, and a good optimizing compiler is required for detecting such loops and replacing them with other machine instructions.

POSSIBLE SOLUTIONS:

Allow slices of multidimensional arrays.

This change would make the following possible:

```
type Index_1 is range ...;
type Index_2 is range ...;

type Matrix is array (Index_1 range <>, Index_2 range <>) of        Float;

M : Matrix(1..6, 1..4);

procedure Invert (M : in out Matrix);

...

Invert(M(1..3, 1..3));   -- invert a submatrix of M

...
```

One problem with this change is that parameters of array types will not always occupy contiguous portions of memory and as a consequence, attributes such as 'Size and 'Address will not make sense anymore.

SUBARRAY SELECTION OF MULTI-DIMENSIONAL ARRAYS

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)

ANSI/MIL-STD-1815A REFERENCE: 4.1.2

PROBLEM:

Current Ada allow the slicing of one-dimensional arrays (i.e. selecting as an object a subvector whose index range is a subrange of the index range of the parent array object), but not of higher dimensional arrays. This is a non-uniformity in the language. It also impacts heavily on the efficiency of many numerical algorithms. Most numerical algorithms which process two-dimensional arrays involve repeated operations on single rows or columns, or (for more modern algorithms aimed at vector and parallel architectures) on square or rectangular submatrices.

The slicing facility (for one-dimensional arrays) can be used for partly copying the contents of an array, like by:

```
A ( 2 .. 5 ) := B ( 4 .. 7 );
```

but also for parameter passing where the formal parameter type is an unconstrained array type, like in:

```
SUM ( X ( L .. U ) )   .
```

An alternative for obtaining a reference to a subarray is a renaming declaration:

```
Y : ARRAY_TYPE renames X ( L .. U ) ;
```

In the latter two cases an object with narrower index constraints is obtained (or passed on as a parameter) from the original object without explicitly making a copy, as is intended, since updates with Y should be carried out in X (L .. U) effectively.
The abilities we require now for multi-dimensional arrays are:

A) to obtain objects of the same dimension with narrowed index constraints, like by:

```
M ( L1 .. U1 , L2 .. U2 )   ,
```

(where M is an object or parameter of a two-dimensional array type)

B) to obtain objects of lower dimension, like extracting a row or column (or a slice of these) of a two-dimensional array object, e.g. to be used as an actual parameter of a call of a subprogram that expects a parameter of a one-dimensional array type. Indicating a row of a two-dimensional array might be provided through:

```
M ( I , L2 .. U2 )
```

which selects the submatrix M (I .. I , L2 .. U2) of case A), but now as an object or parameter of a one-dimensional array type (it provides the I-th row of the matrix M, to be used as a vector).

With these abilities, submatrices can then be treated as single objects; and the operations needed can be coded efficiently, e.g., in assembler if needed. At present, this is not possible in Ada, making the expression of many algorithms much less elegant AND much less efficient. Providing this "subarray selection" might cause additional overhead for its users, but the same users may gain efficiency by simpler subscripting.

IMPORTANCE: IMPORTANT

This requirement, though not stated to be essential in the sense of the word 'ESSENTIAL' as defined in the format, is actually ESSENTIAL for all implementors of Numerically Intensive Computing (NIC) methods. Together with other Ada features like the possibility to declare operators for vector/matrix arithmetic, the facility will be massively used, whereas at present these implementors are essentially prohibited of using the highly useful BLAS (Basis Linear Algebra Subroutines) in Ada.

CURRENT WORKAROUNDS:

By explicit procedures for copying indicated parts of multi- dimensional arrays. This copying is of course wasteful of both time and space. Depending of the number of dimensions, if completeness is aimed at, then both the number of subprograms for all cases of subarray selection and the number of parameters will increase rapidly.

POSSIBLE SOLUTIONS:

By allowing a discrete range for each index of an indexed component, with suitable properties for the type of the resulting object.

ACCESSING CHUNKS OF BIT-VECTORS AND BIT-ARRAYS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 4.1.2, 4.3.2, 4.5.1, 13.10.2

SUMMARY:

Bit vectors are an enormously useful and efficient data structure for solving many problems. Ada already provides for the efficient implementation of and, or, xor, and not as bit-parallel operations on bit-vectors. However, other operations also need to be performed efficiently, and Ada provides no mechanism to implement this performance in a portable manner. An example is the "find-first-bit" function, for which there are either hardware instructions, or at least far better software algorithms, than the obvious bit-after-bit loop. There are needs to be a mechanism to acquire "chunks" of bit-vectors and operate on these chunks.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Bit-vectors used as unordered sets are supported by Ada, but other uses of bit-vectors are not. For example, a hardware priority encoder looks at a set of *n* input lines and returns the index of the highest priority line which is active. For many reasons, one may wish to implement this functionality in software instead of hardware, but keep it reasonably fast. However, Ada does not portably support any mechanisms for doing this. The only truly portable technique for solving this "priority encoder" or "find-first-bit" problem is to loop through the bits one by one looking for the highest priority one that is on.

Yet embedded systems programmers have efficiently dealt with this problem in the past by examining chunks of a bit-vector which are larger than one bit. Only when a non-zero chunk is located is any further processing required, and even then a single table lookup will suffice. This technique is impossible to program in portable Ada.

One possible mechanism for solving this problem would be an operation that operated on a bit-vector (as a whole) and converted it into an integer. Using this operation in conjunction with Ada's current ability to select a subsequence from a bit-vector would allow the extraction of any particular chunk. If compilers recognized this combination of selection and conversion, and if the programmer worked with certain particularly efficient classes of subsequences, then these operations could be efficiently open-coded as byte or word accesses to the underlying bits of the bit-vector.

The nature of the bit-vector-to-integer conversion then becomes the issue. In order to assure portability, the ordering of the bits in a bit-vector relative to the ordering of the bits in an integer then becomes visible. (LRM 13.4 para.5 states that the ordering of the bits within a storage unit is machine-dependent.) Remember, however, that the alternative, is to go completely outside of Ada or use some form of unchecked

conversion, which would be worse.

We suggest that Ada provide a particular definition of bit-vector-to-integer conversion--namely, the most obvious one, that of interpreting the bits as bits in a binary integer, where bv(i-bv'FIRST) becomes the coefficient of 2**i in the integer. The integer produced in this manner has type universal_integer and obvious optimizations exist to avoid multiple-precision arithmetic.

CURRENT WORKAROUNDS:

The only efficient workarounds involve bit-diddling behind Ada's back, which is presumably not very portable or maintainable.

There is nothing in Steelman which would lead one to believe that an explicit conversion between bit-vectors and integers would cause immediate damnation to hacker hell. In fact, one could (liberally) interpret requirement 8F to actually require such a conversion capability.

NON-SUPPORT IMPACT: Continued bit-twiddling.

POSSIBLE SOLUTIONS: Given above.

DIFFICULTIES TO BE CONSIDERED:

Some existing implementations may store bit-vectors with bits in a different order than the bits in an integer. This may require translation, but even reversing the bits in a byte or word is not that difficult--it can be done in one ore more table-lookups. The best situation would be if the bit-ordering were the same.

REFERENCES:

Baker, Henry G. "Efficient Implementation of Bit-vector Operations in Common Lisp." Internal Memorandum, Nimble Computer Corporation, October 1989.

SELECTIVE USE CLAUSES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 4.1.3(14-15), 8.4(2)

PROBLEM:

Currently, USE clauses are an all-or-nothing construct; often a programmer wants direct visibility to some subset of the contents of a package spec (often just the operators such as "=" and "*") without polluting the name space with the entire contents of the package spec: USE clauses are too big a mallet to provide this degree of selectivity.

IMPORTANCE: ESSENTIAL

Workarounds for this are clumsy.

CURRENT WORKAROUNDS:

Renaming provides some relief from this problem; for example, one common use of renaming is to write a "renaming package" that renames the operators from some other package: the idea is that the renaming package is WITHed and USEd, giving direct visibility to the operators without giving direct visibility to anything else. Unfortunately, this is clumsy, requires two packages where one should be sufficient, and only works when programmers are diligent and very knowledgeable.

POSSIBLE SOLUTIONS:

Extend the syntax of the USE clause to support selective using (similar to Modula-2):

* To use a selected component of a package:

```
with Some_Package;
use Some_Package.Some_Entity;
```

* To use all of the operators from a package:

```
with Some_Package;
use Some_Package'Operators;
```

* To use all of the operators for a particular type from a package:

```
with Some_Package;
use Some_Package.Some_Type'Operators;
```

The above would be additive, so that each successive USE would have the effect of adding direct visibility to anything to which direct visibility has not already been granted by a previous USE clause.

COMPATIBILITY:

The proposed solution is quasi-compatible. All previously-compiled code will re-compile successfully unless it contains an implementation-defined'OPERATORS attribute for packages (this is not, however, all that likely).

ALLOW PREFIX OF A NAME TO DENOTE A RENAME OF AN ENCLOSING CONSTRUCT

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 4.1.3(18)

PROBLEM:

Section 4.1.3(18) states that "A name declared by a renaming declaration is not allowed as the prefix". This restriction should be removed. The restriction doesn't introduce any major difficulties, but it does get in the way once in a while and it appears to serve no useful purpose other than to help in disambiguating pathological cases such as

```
type R is
        record
                X : Integer;
        end record;

function F return R is
        function G return R renames F;
        X : Integer;
        begin
                return R' (X => F.X + G.X);
                --F.X. refers to the local variable X;
                --G.X. refers to the X component of the result of a
                --recursive call.
        end F;
```

If the restriction were removed, then presumably 4.1.3(19) would apply to this example and the selected component G.X would then refer to the local variable X.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

USER-DEFINED ATTRIBUTES

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 4.1.4, 7.4.2.

PROBLEM:

It is currently impossible to design software (abstract data) types which, for example, mimic the semantic properties of pre-defined types because of the impossibility of making language attributes applicable to new, private types designed to have similar properties to pre-defined types.

This means that reusable software which is designed to be portable for different precisions and ranges of arithmetic types (real or integer) must use function calls instead of attributes. Hence, functions must also be defined (and used) for the pre-defined types with a possible loss of efficiency.

Example:

A generic package, based on a generic parameter that is a floating-point type (type REAL is digits <>;), can be implemented with a package body in which the pre-defined arithmetic operations ("+", "*", etc.) can be used, but also the attributes that characterize the floating-point model and the machine representation.

Although a user can design a (software) floating-point type with its inherent arithmetic operations and satisfying a floating-point model in the same way as the pre-defined floating-point type(s), such a type cannot be used as the actual generic parameter of the above described generic package.

IMPORTANCE: IMPORTANT

Attributes encourage portability, this is lost if attributes cannot be used by reusable software. This problem will be further exacerbated if additional attributes are defined at 9X to provide additional functionality (for example 'MIN(A,B) 'MAX(A,B)).

CURRENT WORKAROUNDS:

Attribute-yielding expressions are replaced by function calls.

POSSIBLE SOLUTIONS:

Introduce the declaration of attributes, and allow such attributes to be (explicitly) imported as generic

parameters.

ATTRIBUTES CANNOT BE DEFINED WITH RESPECT TO A USER-DEFINED TYPE

DATE: October 14, 1989

NAME: William Thomas Wolfe

ADDRESS: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: (803) 656-2847
E-mail : wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 4.1.4 (4)

PROBLEM:

Attributes can be defined by the Ada language and by an implementation, but cannot be defined with respect to a user-defined type.

IMPORTANCE:

CONSEQUENCES:

Inconsistency between predefined types and user-defined types.

Although it is possible to define procedures and functions over user-defined types, it is not possible to invoke them using the attribute notation. This creates syntactic inconsistency; code must be written one way if an attribute for a predefined type is being used, and a different way if a "synthetic" attribute for a user-defined type is being used.

CURRENT WORKAROUNDS:

Use "synthetic" attributes, as described above.

POSSIBLE SOLUTIONS:

Permit the expression of attributes for user-defined types such that operations can be invoked using the object-oriented attribute syntax, whereby the object itself is an implicitly supplied parameter to the invoked operation.

USER-DEFINED ATTRIBUTES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 4.1.4 (4), Appendix F

PROBLEM:

Implementation-defined attributes that are not supported by some other implementation introduce a severe portability problem that is far worse than that of unsupported rep-specs and pragmas: at least with unsupported rep-specs and pragmas the code will still compile (since the unsupported rep-specs and pragmas are simply ignored), if not execute. Unsupported implementation-defined attributes, on the other hand, cannot just be ignored: so compilation fails. Worst of all, there is no way to simulate/stub unsupported implementation-defined attributes: the code actually has to be modified in order for it to compile.

IMPORTANCE: ESSENTIAL

This revision request fixes a serious portability problem which should not be present in a language designed from the start to be portable; this portability problem is severe enough that if this revision request (or one similar to it) is not folded into the language, then implementation-defined attributes should be forbidden in the next version of the language, regardless of the impact on existing code and compilers.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

It should be possible for the user to define attributes. there are several ways this could be done, one of which is suggested below:

```
type Some_Type is private;

attribute Some_Type'Image
            (This_Instance_Of_Type : in Some_Type) return String;
```

The visibility rules for user-defined attributes should be the same as for subprograms: this would allow user-defined attributes to hide both standard and implementation-defined attributes, if desired.

Note: Not only does this revision request provide a way to fix the portability problem introduced by implementation-defined attributes, but it also makes the language considerably more symmetrical and much more supportive of object-oriented programming. For example, instead of having to define a bunch of hokey attribute-like functions (e.g. Image and Value) for a new type, a programmer would define a set of actual attributes (e.g. 'Image and 'Value) for the new type. This would make user-defined types indistinguishable from pre-defined types, a very desirable feature.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will behave identically during execution except for possible small changes in execution speed.

WHEN A CONSTRAINT ERROR IS TO BE RAISED

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 4.1 #10, 13.7.2

PROBLEM:

A constraint error is to be raised on access of a Null item. It appears that Constraint_error has too many situations that it can be raised therefore losing clarity as to the error cause. Therefore, Constraint_error should only be raised on a violation of a constraint provided in a user defined object. Any other situation gives rise to exploring language situations where the violation could have occurred. For some of those occasions, other exceptions are also associated with the violation. One exception raised for a particular offense should be sufficient.

IMPORTANCE: IMPORTANT

Moderate impact for very large scale programs to determine where to handle the exceptions and the "side effects" for not handling exceptions. The language is so complex that those determinations are not always clear. It causes great maintenance problems to have to put "when others" indications with program IDs to determine where the problem occurred. For the static cases, constraint errors should be recognized early in the compilation process. Range checking has excessive overhead for realtime processes and the developers rely on hardware interrupts for machine processing faults.

CURRENT WORKAROUNDS:

The developer depends on an over-utilization of "unchecked conversions" or the exception handler "when others" null or set a global ID. The clarity of what occurred is really lost.

POSSIBLE SOLUTIONS:

Restrict constraint errors to the situations where constraints have been declared. Attempt to separate the meanings of Numeric_error, Program_error, and Constraint_error so that only one possible situation can be raised. Provide a mechanism so that the user can tell where the error was raised if the exception is propagated or handled by an interrupt handler. If more than one error is allowed for a given situation, define the precedence/priority.

DISCRIMINANTS APPEAR LIKE VARIABLES

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 4.3

PROBLEM:

Discriminants have a major impact on the results of a program unit compilation. They should not be allowed to masquerade as mere objects, or variables. Each discriminant should stand out vividly in the source text. Possibly it should be shown as an attribute or characteristic of an object, using 'DISCRIMINANT_NAME as a tag instead of merely writing DISCRIMINANT_NAME:=some expression; and having an object be elaborated in a surprising way.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Using programming standards to use special naming conventions for "important objects" that can change the outcome of a program or a structure where the object may appear to be one thing that it really isn't.

POSSIBLE SOLUTIONS:

1. Special characters for naming discriminants.

2. Treat a discriminated object with an attribute or tag to make it more readily identifiable.

3. Remove anything that might make a discriminant be confused with an array or a function or even a simple assignment.

NON-STATIC DISCRIMINANTS IN VARIANT RECORD AGGREGATES

DATE: September 14, 1989

NAME: Randall Brukardt

ADDRESS: R.R. Software, Inc.
P.O. Box 1512
Madison WI 53704

TELEPHONE: (608) 244-6436

ANSI/MIL-STD-1815A REFERENCE: 4.3.1(2)

PROBLEM:

Discriminants that govern variant parts in record aggregates are currently required to be static. This can make it difficult to construct values for types for which discriminants may have many values. This requirement is especially onerous when many different discriminant values select the same variant fields.

The Ada standard currently requires a complete record assignment to change a discriminant value. There are only three ways to construct a completely initialized record at once. First, one could assign from an existing initialized record, but this leads to the chicken-or-egg problem: how does one get the first such record? Second, one could use default component values, if they exist and are exactly what is desired; frequently, however, these conditions do not hold, and adding correct default component values may violate other aspects of the program design. Thus, in many cases, one is left with the third alternative: using a record aggregate. However, the current standard allows the declaration of record types for which it is nearly impossible to construct all reasonable record aggregates, simply because of the number of aggregates necessary given the staticness rule. This means that some reasonable data structure choices have to be rejected simply because the language does not support the easy creation of values for the type.

For example:

```
type SWITCHING_METHOD_TYPE is (MANUAL, AUTOMATIC);

type PHONES (NUMBER_OF_LINES : NATURAL := 0) is record
   case NUMBER_OF_LINES is
      when 0 =>
         null;
      when 1 =>
         CALL_WAITING : BOOLEAN;
      when 2 .. NATURAL'LAST =>
         LINE_SWITCHING_METHOD : SWITCHING_METHOD_TYPE;
   end case;
end record;

function MAKE_PHONES (LINES : in NATURAL) return PHONES is
   -- Set up a default PHONES record for the indicated number of lines.
begin
   case LINES is
```

```
            when 0 =>
               return (NUMBER_OF_LINES => 0);
            when 1 =>
               return (NUMBER_OF_LINES => 1, CALL_WAITING => FALSE);
            when others =>
               return (NUMBER_OF_LINES => LINES, -- Currently illegal!
                       LINE_SWITCHING_METHOD => MANUAL);
         end case;
      end MAKE_PHONES;
```

IMPORTANCE: IMPORTANT

Without the change, considerable code space and programmer time will be wasted writing many aggregates or restructuring types.

CURRENT WORKAROUNDS:

Three workarounds are possible.

If the number of possible values of the discriminant is reasonably manageable, the programmer can create one aggregate for each possible discriminant value. This can be wasteful of space if several of the discriminant values share the same fields.

Alternatively, the programmer can create a special enumeration type to be used solely for the discriminant. For the above example,

```
type DISC is (ZERO, ONE, MANY);
```

would work. The new type could be used in place of the intended discriminant. This is wasteful in space, and obscures the design of the data type: some information is duplicated via storage in more than one place.

Finally, a temporary object can be declared with the appropriate discriminants. Then the individual fields can be set. This would look like:

```
function MAKE_PHONES (LINES : in NATURAL) return PHONES is
   -- Set up a default PHONES record for the indicated number of lines.
   RESULT : PHONES(NUMBER_OF_LINES => LINES);
begin
   case LINES is
      when 0 =>
         null;
      when 1 =>
         RESULT.CALL_WAITING := FALSE;
      when others =>
         RESULT.LINE_SWITCHING_METHOD := MANUAL;
   end case;
   return RESULT;
end MAKE_PHONES;
```

While this solution works well for this particular example, in general it leads to verbose and un-Ada-like solutions where many fields are set one at a time.

POSSIBLE SOLUTIONS:

Remove 4.3.1(2).

Ada compilers already must generate code to check that record fields within a variant exist (to implement 4.1.3(8)), so removing this requirement will not add significant work for compiler writers.

It would make sense to add one restriction in place of 4.3.1(2). It should be required that there be no OTHERS choice in a record aggregate if any discriminant that governs a variant part has a non-static value specified in that aggregate. This prevents pathological cases in which it is not clear until run time which fields are indicated by OTHERS.

At run time, the exception CONSTRAINT_ERROR would be raised if the values given to the discriminants did not match the components given in the record value.

An alternative solution is to allow partial aggregates and partial aggregate assignment. This would make the last work-around much more workable, but at a major cost in language and compiler complexity.

INCONSISTENT TREATMENT OF ARRAY CONSTRAINT CHECKING

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 4.3.1(3), 5.2(3), 5.2.1, 5.8(6), 6.4.1(6), 6.4.1(7)

PROBLEM:

Given the following declarations:

```
subtype Short_String_Subtype is String (1 . 5);

subtype String_Size_Subtype is Natural range 0 .. 80;

type Resizeable_String_Type (Size: String_Size_Subtype := 0)is
        begin
    Value:      Vector_Type (1 .. Size)
        end record;

X, Y:  Natural          -- X & Y are set at run-time

Some_Text: String (X .. Y);      -- initialized at run-time

A_Resizable_String:     Resizeable_String_Type;
```

According to RM paragraph 4.3.1(3) the following aggregate will cause a Constraint_Error because the expression Some_Text does not "belong to the subtype of" A_Resizable_String.Value.

```
A_Resizable_String := (Size => Some_Text'Length,
                       Value => Some_Text);
```

However, if I set the A_Resizable_String.Size to the value of Some_Text'Length by the following assignment statement:

```
A_Resizable_String :=  (Size => Some_Text Length,
                        Value  => (1 .. Some_Text'Length => ' '));
```

then according RM paragraph 5.2(3), I can set A_Resizable_String.Value to the value of Some_Text with assignments such as the following, because "the assignment involves a subtype conversion as described in section 5.2.1".

A_Resizable_String.Value := Some_Text;

As a result of the above rules, assigning an aggregate to a record is NOT equivalent to assigning the components of the record individually (which may be difficult if there is a need to change a discriminant as shown in the example above).

The rules for array constraint checking during parameter association in procedure calls [6.4.1(6) and 6.4.1(7)], and the expression in a return statement [5.8(6)] are identical to those for aggregates.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The most "elegant" workaround is explicit subtype conversion. For example: "replace" the assignment of the record aggregate shown above, with the following block.

```
declare
        subtype Value_Subtype is String (1 .. Some_Text'Length);
begin
        A_Resizable_String := (Size     => Some_Text'Length,
                Value => Value_Subtype'(Some_Text));
end;
```

However, this technique can be awkward since it may require creation of additional subtypes at run-time as in the example shown here.

POSSIBLE SOLUTIONS:

Change paragraphs 4.3.1(3), 5.8(6), 6.4.1(6), and 6.4.1(7) to match the wording of paragraph 5.2(3) which provides for array subtype conversion.

UNRESTRICTED COMPONENT ASSOCIATIONS

DATE: August 2, 1989

NAME: James W. McKelvey

ADDRESS: R & D Associates
P.O. Box 5158
Pasadena, CA 91107

TELEPHONE: (818) 397-7246

ANSI/MIL-STD-1815A REFERENCE: 4.3.1, 3, 4.3.2, 11

PROBLEM:

Component associations are more restrictive than normal assignments.

Example:

```
X : String(1 .. 5) := "ABCDE";

type Test_Type
is
record
  Component : String(1 .. 2);
end record;

Y : Test_Type := Test_Type'(Component => X(4 .. 5));
```

The last line will raise Constraint_Error because of (4.3.1, 3). However, an assignment like:
Y.Component := X is acceptable.

A single aggregate assignment often replaces a series of subcomponent assignments. This is desirable practice primarily because the aggregate must be complete, if the record type changes in some way, compilation errors will target the aggregate for modification. No such safety valve exists for individual subcomponent assignments. It is therefore advantageous for an aggregate to be as much like a series of assignments as possible.

Similar arguments apply to array aggregates.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Replace the aggregate with a series of subcomponent assignments, or, preferably, add explicit subtype conversions.

POSSIBLE SOLUTIONS:

Modify the third sentence in (4.3.1, 3) to read as follows:

A check is made that the value of each subcomponent of the aggregate belongs to the subtype of this subcomponent, except in the case of a subcomponent that is an array (the association then involves a subtype conversion as described in section 5.2.1).

This borrows language from (5.2, 3). A similar change would also be made to (4.3.2, 11).

OTHERS CHOICES IN ARRAY AGGREGATES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 4.3.2(3,6)

PROBLEM:

The restrictions on array aggregates given in 4.3.2(6) should be removed. For example, the following should be legal:

```
X: String ( 1..10) := (1 => 'x', others => 'y');
```

On the other hand, 4.3.2(6)'s definition of a static OTHERS choice should be tightened up slightly, perhaps by adding the text "and all other choices are compatible with the applicable index constraint" to the end of the sentence. For example, the following should not be legal:

```
subtype S is String (1 .. 3);
Y : S := S'(999 => 'x', others => 'y');
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use named subtypes and qualified expressions.

POSSIBLE SOLUTIONS:

BLANK PADDING FOR STRING ASSIGNMENTS

DATE: August 30, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 4.3.2(4), 5.2.1(2)

PROBLEM:

Ada does not automatically pad string values with blanks when they are assigned to strings. If the value is shorter then the variable on the left-hand-side of the assignment operator, CONSTRAINT_ERROR is raised. Due to language restrictions, blank padding cannot be forced with either default values or an others choice.

IMPORTANCE: ESSENTIAL

Without some mechanism to handle this, string handling in Ada will continue to be hard to read and error-prone. For applications requiring string handling, other languages will be used in preference to Ada.

CURRENT WORKAROUNDS:

Unconstrained string types. This can be an extremely expensive solution. Explicit slices. This converts a simple string concatenation to a "mare's nest" of unreadable and unmaintainable code.

POSSIBLE SOLUTIONS:

Allow an array aggregate with an unconstrained others choice as the final concatenation of an array expression, provided that it occurs as the value of an assignment or a component of a qualified expression.

Allow assignment of an array value that is shorter than the variable of the assignment, provided that the variable was defined with an others choice for its initial value.

As above, but limited to strings.

Treat string as a separate data type, rather than as an unconstrained array, and define assignment and equality to include blank padding. Continue to allow concatenation, slicing, etc. on strings, with the same syntax as for arrays overload assignment and equality as above, but continue to treat strings as arrays.

OTHER CLAUSES IN ARRAY AGGREGATES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 4.3.2(4-8)

PROBLEM:

LRM 4.3.2, paragraphs 4-8 spell out in excruciating detail the cases in which an OTHERS clause is and is not permitted in an array aggregate. The rationale for these restrictions is not given, and they all seem arbitrary. Almost nobody correctly understands and/or memorizes the various restrictions.

One small example of the capriciousness of this part of the standard:

* This complies:

```
type Bar is array (1 .. 10) of Integer;
Boo : Bar := Bar' (1 => 8, 5 => 9, others => 0)
```

* This doesn't:[1]

```
type Bar is array (1 .. 10) of Integer;
Boo : Bar := (1 => 8, 5 => 9, others => 0)
```

IMPORTANCE: ESSENTIAL

This area of the standard is maddening and need extensive help.

CURRENT WORKAROUNDS:

Never try to use OTHERS clauses an array aggregates. Try to use OTHERS clauses in array aggregates an hope for the best: when they don't work scratch your head in wonderment and try some other approach.

POSSIBLE SOLUTIONS:

Clean up this part of the standard so that OTHERS clauses either always work, or fail to work in only the rarest occasions for the clearest of reasons. At a minimum they should work as well as OTHERS clauses do for record types.

---

[1] Can you explain why?

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

AGGREGATES ARE DIFFICULT TO READ WHEN NESTED IN SUBPROGRAM CALLS AND/OR CONTAINING NESTED PARENTHESISED EXPRESSIONS

DATE: June 7, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 4.3(2)

PROBLEM:

Aggregates are difficult to read when nested in subprogram calls and/or containing nested parenthesised expressions. A conflict with readability as a design goal.

IMPORTANCE:

This issue is somewhat important.

CURRENT WORKAROUNDS: None (except for APSE or preprocessors, as always)

POSSIBLE SOLUTIONS:

Readability/writability would be enhanced if aggregates could be written as '[' component_associations { ',' component_association} ']'.

Certainly in nested expression that tend to contain a lot of nested parentheses, brackets would be more visible.

The rule 04.03(04) that imposes named notation for single component aggregates to prevent ambiguity with the nested expressions is a direct consequence of using parenthesis instead of brackets.

The only reasonable argument against brackets is probably that they are not available on all keyboards. Such problems however should not deteriorate the language quality on the majority of the systems that directly support these characters, and hence replacement characters would be a much better solution. So, it would be desirable to have the aggregate syntax described using brackets, and allowing parentheses as replacement character with the restriction of 04.03(04) when parentheses are used.

This would not invalidate any existing Ada source, and allow production of a more readable source using Ada9X.

Portability is less of an issue here. The machines with keyboards without brackets do support brackets in

ascii files, they are just hard to type in. In other words, porting is still no problem (or still the same problem).

AGGREGATES FOR SINGLE-COMPONENT COMPOSITE TYPES

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 4.3(4)

PROBLEM:

Currently, if an array or a record has only one component, it can be given an aggregate assignment by using named notation only. This seems to be an unnecessary exception to the general rule that named and positional notations can be used interchangeably.

Examples:

```
rec := (value);
ary := (value);
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Used named notation.

POSSIBLE SOLUTIONS:

SHORT CIRCUIT

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 4.4, 4.5.1, 4.5.2

PROBLEM:

Short circuit semantics are unnecessarily prohibitive and add much verbosity/inefficiency in writing Ada code. The AND THEN and OR ELSE are just not necessary in modern optimizing compilers. Code generation efficiency is obtained for embedded systems which may have more than one variant module. It is desirable to short circuit as soon as the compiled result recognizes that the entire path does not need to be evaluated. This removes the chance that exceptions must be handled for meaningless paths before a branch is taken. The optimization wording in the LRM seems to allow conditional compilation but not full short-circuiting. If the results cannot be tested and many of the evaluations are implementation dependent, then the LRM should specify left-to-right order and be silent on the code generation method.

IMPORTANCE: IMPORTANT

High for embedded systems.

CURRENT WORKAROUNDS:

Not many to overcome the LRM for full short circuit and almost no vendor will support conditional compilation under the current validation process.

POSSIBLE SOLUTIONS:

1. Add semantics to allow the short circuit to occur as soon as the system can recognize the result without evaluating the entire boolean expression. Also, allow constant boolean expressions to be used as conditional compilation (for greater reuse) where no code has to be generated from unreachable paths. A "warning" should be issued--see write up on levels of errors, reference #0-028.

2. Remove AND THEN and OR ELSE as unnecessary in the language and add semantics so that the system can short circuit as soon as the result is recognized.

OVERLOADING

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 4.4, 6.1, 6.6, 6.7

PROBLEM:

Currently, testing and maintenance is the predominant activity in the software life cycle. The LRM allows overloading which tends to obfuscate source code and is not in the spirit of more reliable, easily maintainable code.

For example, if + and - are already defined operators for a given type then do not allow the user to overload the same type with an operator that changes its meaning, i.e., - cannot mean +. Force the user to adopt another symbol, after all Ada supports some 10 other symbols that the user can "overload".

The language should provide semantics to forbid overloading a symbol with another one of the same type. Overloading should not be allowed for inequality in subpara. 4 of 6.7.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Programming standards.

POSSIBLE SOLUTIONS:

<<minor impact>> Add semantics to subpara. 4 of 6.7 to include multiple symbol definitions for the same type objects, e.g., +,-,*,/ cannot be reused for overloading the same numeric operator.

NUMERIC OPERATORS FLOOR, CEILING NOT PREDEFINED

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 4.4

PROBLEM:

Numeric operators FLOOR, CEILING not predefined.

Why did the language take so good care of the difference between mod and rem (which could also have been left to the programmer) while the treatment of the (still rather classic) CEILING and FLOOR explicit conversions was not supported? It would be no extravagant luxury to have a CEILING and FLOOR conversion in addition to the numeric conversion that rounds indeterministically for INTEGER + 0.5.

As a remark, the language definition itself needs the CEILING function to define the correspondence between the number of digits and the number of bits of a floating point number (ARM. 03.05.07(06)), but there one prefers an informal definition: "the integer next above..."

As a more practical example if one wants to calculate the exact number of digits needed for an exponent to output a number as 0.NDDD E xxx, then the exponent can be easily expressed as CEILING(LOG10(NUMBER)). Currently one would be tempted to write INTEGER(LOG10(NUMBER)) = 0, and the exponent is given by INTEGER(0.5) which might be 0 as well as 1.

So, in general one should write a function, for example:

```
generic
        type T_INTEGER is range<>;
        type T_FLOAT is digits<>;
function CEILING (F:T_FLOAT) return T_INTEGER;
function CEILING (F:T_FLOAT) return T_INTEGER is
        ALMOST_CEILING:constant T_INTEGER:=T_INTEGER (F+0.5);
begin
        if F= T_FLOAT (ALMOST_CEILING +1)
        then---T_INTEGER(F) + 0.5 has been rounded upward to T_INTEGER (F) + 1
                return ALMOST_CEILING + 1;
        else return ALMOST_CEILING;
        end if;
end;
```

Every user will probably agree that this job should be done by the compiler suppliers, because they know

how they round at integer +0.5. But the Ada definition only allows the solution presented above, a generic unit to be instantiated by the users. This has several drawbacks, the need to instantiated it whenever it is needed is one of them, and lack of portability since the feature is not in the standard environment.

IMPORTANCE: IMPORTANT

To avoid pitfall with round-off explained above.

CURRENT WORKAROUNDS: Add to the development work.

POSSIBLE SOLUTIONS:

The obvious solution is adding CEILING and FLOOR declarations to package STANDARD.

The problem with this approach is that they take any real number as argument, and hence they should be defined for each possible combination (predefined_integer_type, predefined_real_type). Hence they should be declared implicitly (as currently done for "*" and "/"):

```
--function CEILING (RIGHT : universal_real) return universal_integer;
--function FLOOR (RIGHT : universal_real) return universal_integer;
```

Alternatively, they could be introduced as additional attributes CEILING and FLOOR for all integer types which are functions that convert a universal_real value to the next higher or next smaller integer.

The problem could be partially solved if the language standard would require either upward or downward rounding for half integers, such that CEILING and FLOOR are easier to write. But this possibility was probably rejected a long time ago.

SCALAR OPERATORS MIN, MAX NOT PREDEFINED

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 4.4

PROBLEM:

Scalar operators MIN, MAX not predefined.

As a result many programmers end up stuffing their code with local definitions of functions MIN and MAX.

The key problem is obviously not only to implement these functions, but to find a proper place to declare them, and this is not so easy.

Functions as compilation units need to be with-ed function by function and cannot overload each other, so one cannot declare MIN and MAX for each predefined scalar type.

Placing such functions in generic units implies an instantiation at all places of use which is a huge job for the compiler supplies these functions in STANDARD.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Add to development work.

POSSIBLE SOLUTIONS:

The obvious solution is adding MIN and MAX declarations to package STANDARD.

```
function MIN (LEFT, RIGHT :INTEGER) return INTEGER;
function MAX (LEFT, RIGHT :INTEGER) return INTEGER;

function MIN (LEFT, RIGHT :REAL) return REAL;
function MAX (LEFT, RIGHT :REAL) return REAL;

--function MIN (LEFT, RIGHT : any_fixed_point_type) return same_fixed_point_type;
--function MAX (LEFT, RIGHT : any_fixed_point_type) return same_fixed_point_type;
```

In the Ada Europe meeting variable argument lists were mentioned as a possible problem with MIN and MAX support. This problem could be separated from this issue by remarking that there is an equal need for variable parameter lists for the "+" operation, or it could be solved by stating that MIN and MAX are also defined for any one-dimensional array type with scalar component subtype.

For example:

```
function MIN (RIGHT : STRING) return CHARACTER;
function MAX (RIGHT : STRING) return CHARACTER;
```

LOGICAL OPERATIONS ON ADA INTEGERS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 4.5

SUMMARY:

Ada does not currently support "logical" operations on integers such as "logical AND", "logical OR", "logical XOR', which are commonly and efficiently available in virtually every other low and high level computer language today. While the lack of these operations is considered essential from the standpoint of "data abstraction", this standpoint does not take into account the tremendous inconvenience involved in translating programs from other languages. We recommend the inclusion of these operations--at least as "second class citizens"--through their incorporation as a required package in the language having a common, portable interface.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The availability of traditional logical operations on integers in virtually all other low-level and high-level languages bears testament to their usefulness. Almost all Ada implementation offer some sort of machine-specific provisions for these operations, but due to the lack of a standard, there is no portability among implementations.

The lack of logical operation in Ada has been justified on the "religious" grounds that Ada integers are the abstract mathematical notation of integers, and any operations on integers that look at a binary representation of an integer is considered too representation dependent. However, all modern computers use the binary representation for integers, and almost all utilize the 2's complement representation for negative integers. As a result, the efficient implementation of logical operations is quite feasible on all modern computers--especially the sort that are being ;utilized for "mission-critical" tasks for which Ada is required.

The specific recommendation is that logical operations on integers be supported, where the definition of the result is as if the representation were in 2's complement notation. Any result which lies outside the range constraint would cause an exception, unless this were suppressed.

There are two levels of recommendation, minimal and maximal.

Minimal recommendation--the logical functions LOGAND, LOGOR, LOGXOR and LOGNOT be support for integers. Any of the other 12 two-argument Boolean functions can be constructed from these.

Maximal recommendation-- the function BOOLE(code,operand1, operand2) be supported for integers, and

the functions LOGAND, LOGOR, LOGXOR and LOGNOT be supported as special cases of BOOLE. The "code" operand to BOOLE is an integer 0..15 which represents which of the 16 possible Boolean functions of 2 bits is being requested. This function is supported as a single instruction in many RISC, CISC and micro-coded processors.

CURRENT WORKAROUNDS:

Even people who agree with the religious argument for not including Boolean functions in Ada sometimes have to convert programs from other languages which utilize them, and the costs in terms of rewriting, inefficient code, and non-portability far exceeds the advantages of the religious approach.

Some examples:

(LOGAND 2 3) = 2
(LOGAND -8 2) = 0
(LOGAND -8 16) = 16
(LOGAND -1 x) = x
(LOGOR 0 x) = x

LOGNOT(x) is simply "-x-1", although many computers can perform this operation slightly faster as a true Boolean Operation.

LOGAND(x,y) is very difficult to simulate in general, although some cases can be simulated through other means--e.g., LOGAND(x,65535)=MOD(x,65536), which is probably slower on most Ada implementations. The same reasoning applies to LOGOR. LOGOR can be computed from LOGAND by using DeMorgan's law: LOGOR(x,y) = LOGNOT(LOGNOT(x),LOGNOT(y)) = -LOGAND(-x-1,-y-1)-1.

LOGXOR(x,y) is likewise difficult to simulate in general, but it can be simulated if LOGAND is available through the following identity: LOGXOR(x,y)=(x+y)-2*LOGAND(x,y).

```
LOGNOT(x)=  -x-1
LOGOR(x,y)=  LOGNOT(LOGAND(LOGNOT(x),LOGNOT(y))
                    -LOGAND(-x-1,-y-1)-1
LOGXOR(x,y)=        (x+y)-2*LOGAND(x,y)
```

NON-SUPPORT IMPACT:

The current lack of productivity of conversion of programs from other languages into Ada will continue. The lack of portability of the resulting programs will cause additional loss of productivity during the entire lifecycle.

POSSIBLE SOLUTIONS:

The only reasonable solution is the generation of the machine code that exists on all modern computers.

One other approach would be to allow the conversion of bit-vectors to and from integers in a portable way. This approach is taken up in another request entitled "Accessing chunks of bit-vectors and bit-arrays".

DIFFICULTIES TO BE CONSIDERED:

The only difficulties would be the detection of subrange constraints, which is already being done for normal arithmetic.

REFERENCES:

Schacht, Eric N. "Ada Programming Techniques, Research, and Experiences on a Fast Control Loop System". Proc. of Using Ada: ACM SIGAda Int'l Conf., Boston,MA, Dec. 1987, 164-169. Describes efficiency problems of Ada logical operations on integers, including shift operations.

SHIFT OPERATIONS ON INTEGERS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 4.5

SUMMARY:

Ada should be extended with arithmetic binary shift operations, as all modern computers offer some form of binary shift, and the ubiquity with which these operations appear in virtually every other modern lower-level or higher-level computer language attests to their usefulness. While shift operations have been criticized on "religious" grounds by Ada purists, their lack causes a great lack of productivity in the conversion of programs into Ada, a continuing lack of efficiency, and a lack of portability among implementations.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

We recommend the addition of the single function ASH(x,y) whose definition is equivalent to:

```
ASH(x,y)=   x*(2**), if y > 0
            x, if y = 0
            (x-MOD(x.s**(-y)))/(2**(-y)), if y < 0
```

If y is positive, ASH(x,y) is the traditional "arithmetic left shift" found in most 2's complement computers. If y is negative, ASH(x,y) is the traditional "arithmetic right shift" (by -y) found in most 2's complement computers.

CURRENT WORKAROUNDS:

Implementing certain operations such as "find-first-bit" (e.g. the length) of an integer can be efficiently simulated in software by a number of table lookups by considering portions of the integer--e.g. one byte at a time. However, the speed of these operations depend critically on the ability to shift and extract portions of the integer quickly--i.e., perform logical operations on integers. Portable implementations of these operations which are also quite efficient could be better done if shifts were a standardized part of the Ada language.

The workaround for ASH(x,y) for non-negative y is obvious--use the expression x*(2**y). If compilers are smart enough to recognize this situation and produce the correct machine instruction, this would be acceptable. However, for negative y, the equivalent expression is too complicated and too unreadable to be acceptable.

NON-SUPPORT IMPACT:

Difficulty in translating code into Ada from other languages, inefficient code, and lack of portability of vendor supplied fixes.

POSSIBLE SOLUTIONS:

The only reasonable solution is to have the compiler generate the obvious machine instruction.

One other approach would be to allow the conversion of bit-vestors to and from integers in a portable way. This approach is taken up in another request entitled "Accessing chunks of bit-vectors and bit-arrays".

DIFFICULTIES TO BE CONSIDERED:

The definition of ASH9x,y) is most efficient for 2's complement representations. However, producing efficient code for 1's complement machines is not that difficult, either, because of the following identity:ASH(x,-n)=LOGNOT(ASH(LOGNOT(x),-n)), for n >0.

REFERENCES:

Schadt, Eric N. "Ada Programming Techniques, Research, and Experiences on a Fast Control Loop System". Proc. of Using Ada: ACM SIGAda Int'l Conf., Boston,MA, Dec. 1987, 164-169. Describes efficiency problems of Ada logical operations on integers, including shift operations.

MULTIPLE PRECISION INTEGER OPERATIONS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 4.5

SUMMARY:

While it is possible to implement portable multiple-precision integer algorithms using Ada, the resulting implementations are at usually two or four times slower than they need to be, because Ada does not provide convenient multiple precision arithmetic functions. This proposal attempts to correct this lack of putting back operations which are available in most computer assembly languages, but not in most higher-level languages.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Integer operations of precision greater than that available on many implementations of Ada is often required, even in embedded systems. The most obvious example is that of an Ada compiler, which is required to perform arbitrary precision arithmetic for the implementation of universal_integer, but example also appear in embedded systems for timers, and for implementing calculations which must be extremely accurate (e.g., astronomical calculations).

While a portable implementation can be done, it is usually twice as slow as it should be for addition, and four times as slow as it should be for multiplication, because it cannot take advantage of the hardware capabilities of almost all modern computers for handling multiple precision arithmetic. We propose a solution for this problem which will alleviate this inherent inefficiency.

ADD(x,y,low,high)	x,y,low,high are the same type (INTEGER,LONG_INTEGER,etc.) x,y are IN parameters low, high are OUT parameters low = MOD(x+y,(T'LAST+1) -- the "true" x+y high- (x+y-low)/(T'LAST+1) -- the "true" x+y
MULTIPLY(x,y,low,high)	x,y,low,high are the same type x,y are IN parameters low, high are OUT parameters low = MOD(x*y,(T'LAST+1) -- the "true" x*y high- (x*y-low)/(T'LAST+1) -- the "true" x*y

When using 2's complement arithmetic and T'FIRST=-T'LAST-1, it should be possible to open code these operations.

CURRENT WORKAROUNDS:

The workarounds for multiple precision integer operations are obvious, and have been implemented in many packages. However, the lack of access to the efficiency of the builtin hardware operations will cause some to prefer non-portable solutions involving other languages or machine code.

NON-SUPPORT IMPACT:

Inefficiency in running code, lack of portability, or both.

POSSIBLE SOLUTIONS:

There is probably a more elegant solution for no-symmetric ranges (e.g., for unsigned integers), but I will leave that to the distinguished committee.

DIFFICULTIES TO BE CONSIDERED:

Dealing with other than the operations provided by the hardware will cause some slowdown, but not nearly as much as going to half precision.

AXIOMS TO BE OBEYED BY BUILT-IN OPERATIONS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 4.5.X

SUMMARY:

There are several assumptions built into the Ada language description which should be made explicit. While our proposal involves making the obvious even more obvious, it is still worth clarifying these issues. In particular, the following rules should be obeyed by built-in operations:

For all x,y, "x/=y" is equivalent to "not(x=y)"
For all x,y, "x<=y" is equivalent to "not(x<y) or (x=y)"
For all x,y, "x>=y" is equivalent to "not(x>y) or (x=y)"
For all x,y, "x=y" is equivalent to "not(x/=y)"
For all x,y, "x<y" is equivalent to "not(x>=y)"
For all x,y, "x<=y" is equivalent to "not(x>y)"
For all x,y, "x>y" is equivalent to "not(x<=y)"
For all x,y, "x>=y" is equivalent to "not(x<y)"
For all x,y, "x not in y" is equivalent to "not (x in y)"

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The following rules should not be obeyed by the built-in operations:

For all x,y, "x/=y" is equivalent to "not(x=y)"
For all x,y, "x<=y" is equivalent to "not(x<y) or (x=y)"
For all x,y, "x>=y" is equivalent to "not(x>y) or (x=y)"
For all x,y, "x=y" is equivalent to "not(x/=y)"
For all x,y, "x<y" is equivalent to "not(x>=y)"
For all x,y, "x<=y" is equivalent to "not(x>y)"
For all x,y, "x>y" is equivalent to "not(x<=y)"
For all x,y, "x>=y" is equivalent to "not(x<y)"
For all x,y, "x not in y" is equivalent to "not (x in y)"

CURRENT WORKAROUNDS:

Compiler optimization and program verification/validation is greatly hampered if there are no axioms governing the action of the built-in operations of Ada. While the use of overloading by the programmer can violate the above axioms, that is an issue that affects a particular program or subsystem. However, if

the above axioms cannot be relied upon for all Ada built-in (non-overloaded) operations, then very little optimization or verification can be performed.

NON-SUPPORT IMPACT: Slow and/or buggy programs.

POSSIBLE SOLUTIONS:

Make the above propositions into axioms.

DIFFICULTIES TO BE CONSIDERED:

Most implementations already adhere to these specifications, and those that don't, should.

ADA SHOULD SUPPORT CHECKSUMS IN COMMUNICATION PROTOCOLS

DATE: June 16, 1989

NAME: Ivar Walseth

ADDRESS: Kjell G. Knutsen A/S
Box 113
4520 Sor-Audnedal
NORWAY

TELEPHONE: +47 43 56205

ANSI/MIL-STD-1815A REFERENCE: 4.5

PROBLEM:

Many algorithms require bitwise operations on integers (i.e., checksums in communication protocols). Since such operations are commonly available on most CPUs, Ada should support them.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

1. Use pragma interface to another language. This is inefficient, and work must be done every time the software is ported another CPU.

2. Use representation clause and unchecked conversion. This may be unsafe and it is not supported by all validated compilers.

POSSIBLE SOLUTIONS:

Support the following operators for any integer type (or at least for any unsigned integer type):

```
function "and" (left, right:int)    return int;
function "or"  (left, right:int)    return int;
function "xor" (left, right:int)    return int;
function "not" (operand:int)        return int;
function shift-left (operand:int; bits integer) return int;
function shift-right (operand:int; bits integer) return int;
```

Allow modulo arithmetic by some means. This could be implemented as:

```
type byte is range O..255;
pragma module_arithmetic (byte);
```

REDEFINITION OF ASSIGNMENT AND EQUALITY OPERATORS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 4.5(1-2), 5.2(1), 6.7(4), 7.4.4(3), 7.4.4(3,6,7,9,11)

PROBLEM:

The standard allows operators such as "*" and "+" to be redefined by the programmer, both for predefined types and for new types defined by the programmer. This is very nice, both because it provides the programmer with a great deal of flexibility and because it supports object-oriented programming techniques with a symmetrical language definition.

Unfortunately, the current standard balks at allowing redefinition of assignment (":="), equality ("+"), and inequality ("/=") operators uniformly across all types, and this frustrates and annoys programmers who are just trying to use the language to do object-oriented programming. (Inequality is especially odd because it is always obtained implicitly by negation of the equality operator rather than being treated as a separate operator).

For example, a diligent and well-meaning programmer who has just defined a private type of some kind might very well wish to define a test for equality for the type and call this test for equality "=" (thereby overriding a potentially undesirable default test for equality). Unfortunately, unless the programmer makes the type a limited private type, this is not possible.

Suppose the programmer then does decide to use a limited private type. Now "=" can be defined, but there is no way for ":=" to be defined, which introduces a whole other set of problems.

IMPORTANCE: ESSENTIAL

This whole area of the standard is a minefield because it is inherently asymmetrical for no apparent reason: students go nuts trying to remember when they can and cannot do things that should always be possible to do.

CURRENT WORKAROUNDS:

Write hokey Is_Equal and Is_Not_Equal functions for all but limited private types and hope programmers will remember to use them instead of the default "=" and "/=" tests (which may be totally inappropriate).

Write hokey Assign procedures for limited private types (or use pointers to limited private types to circumvent the lack of ":=").

POSSIBLE SOLUTIONS:

Clean up this area of the standard so that it acts like a real object-oriented language, with proper inheritance and overriding of operators. Make it possible to redefine equality, inequality (independent of equality), and assignment for all types. Make redefinition override the default version of these operations. Make the only difference between limited private and regular private types be the fact that there are fewer predefined operators for limited private types.

COMPATIBILITY:

Believe it or not, the solutions proposed above are upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

PROVIDE SUPPORT FOR CHARACTER COMPARISON BASED ON THE LOCAL ALPHABET

DATE: March 25, 1989

NAME: Erland Sommarskog (Endorsed by Ada in Sweden)

ADDRESS: Erland Sommarskog
ENEA Data AB
Box 232
S-183 23 T[BY ("[" ="A" with dots.)
SWEDEN

TELEPHONE: +46-8-7922500
E-Mail: sommar@enea.se

ANSI/MIL-STD-1815 AREFERENCE: 4.5.2 if any.

PROBLEM:

Comparison of characters and strings today is based on their ASCII codes. This seldom not very meaningful for any other language than English. In many languages the rules cannot even is be described by a simple enumeration rule. For instance in German "A" with dots is sorted under normal "A". The dots only have significance when there is no other difference. Another case is Spanish, where "CH" is considered as a separate letter.

IMPORTANCE: IMPORTANT

While it could be argued that is something to have in a package that is not part of the language, I would like to point to that comparison of strings is a very basic operation.

CURRENT WORKAROUNDS:

Write a package to provide the desired operations.

POSSIBLE SOLUTIONS:

Alternative 1.

> Provide a standard package (like Text_io) with the required operations. (It is the author's feeling that standard packages should be more loosely coupled to the language, for instance being defined in a separate document. I know that there are other revision requests on this issue, so I shall not elaborate this further here.) Beside the comparison operations, the package could also include predefined rules for major languages.

Alternative 2.

> Place the extended comparisons in the predefined STANDARD. The operations for loading and selecting an alphabet (see above) should probably be in a special package within Standard (or

separately) if for nothing else for notational reasons. The default behavior would be comparison on ASCII codes.

MULTIPLYING OPERATOR "/"

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 4.5.5

PROBLEM:

Multiplying operator "/". Isn't this an incoherent approach to avoiding ambiguity? Crystal-clear operations like V_FLOAT := 5 are forbidden because implicit conversion from universal integer to a floating point type is considered dangerous. Of course this would be a dramatic approach for compilers that would not do the conversion, but that reasoning surely does not apply to Ada. So, one chooses to keep a formal difference between two things that are semantically very close (5 and 5.0).

Now, for the division operator, there are two very different operations, integer division and floating point division. Here the language uses a single token, '/' to represent the semantically different operations. Why didn't one follow Pascal here, using operator div for integer division and "/" for floating point division?

In the Ada Europe meeting it was mentioned that in Ada (unlike Pascal) integers were not treated as a subset of reals, but as something different. This is undisputable, but the question is: does support for a predefined operator:

```
function "/" (LEFT,RIGHT : INTEGER) return universal_real;
```

imply that INTEGERs are a subset of REALs? Or does support for an implicit type conversion from universal integer to universal_real imply this?

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS: Introduce operator div.

OPERATORS FOR ALL PREDEFINED INTEGER TYPES, NOT JUST STANDARD.INTEGER

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 4.5.5(7), 4.5.6(5)

PROBLEM:

For an implementation which provides a predefined Long_Integer type, it would be useful if the following assignment statement were legal:

```
        X : Long_Integer := Long_Integer (Integer'Last) + 37;
        Y : Duration := Duration'Delta;
begin
        Y : X*Y;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

This, and analogous problems, could be solved if the operations defined on Standard.Integer in 4.5.5(7) and 4.5.6(5) were also defined on every predefined integer type, including Universal_Integer. It is important that operations on Universal_Integer be provided because otherwise any implementation which provides more than one predefined integer type would then have to reject the following (which is currently legal)

```
        Y : Duration := 1.0;
        begin
        Y := 2*Y;
```

because the call to "*" would be ambiguous.

FIXED MULTIPLICATION & DIVISION WITH UNIVERSAL REAL OPERANDS

DATE: October 4, 1989

NAME: Terence J. Froggatt

DISCLAIMER:

The views expressed in this note are those of the author and do not necessarily represent those of SD-Scion Plc.

ADDRESS: Ada Division, SD-Scion Plc.
Pembroke House, Pembroke Broadway,
Camberley, Surrey, GU15 3XD, U.K.

TELEPHONE: Home: +44 252 613996
Work: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 4.5.5(10)

PROBLEM:

In the present Ada language, a floating point number can be manipulated or divided by a real literal constant (or any universal real named number) whereas a fixed point number cannot be: the programmer has to give the literal (or named number) a type. The need for this facility was recognized in one of the earliest Ada issues, AI-20, for literals, and subsequently in Ai-376 for universal real operands generally.

The reason given for the lack of these literal operations was the uncertainty over the accuracy to which the constant has to be held at run-time, (see Ada Letters IV-2.68 & VI.6-77), when the language was defined in 1983. Since then, the problems have been solved, and it has been shown that the restriction is unnecessary.

(In the currently approved forms of AI-20 and AI-376, it is stated that providing this facility requires the same support from the compiler as would supporting arbitrary 'SMALLs. This is misleading, because only a simple subset is required, as is explained below.)

IMPORTANCE: IMPORTANT

This is an IMPORTANT change, not simply because it adds a needed facility to the set of fixed point operations, but because it will provide greater uniformity amongst real types, and so make the language closer to its original design intentions.

CURRENT WORKAROUNDS:

The usual workaround for this problem is to type-convert the literal or named number. Unfortunately this may reduce its accuracy...

```
        PI: constant := 3.14.....;
begin
```

```
        FIXED_VALUE := FIXED_TYPE (FIXED_VALUE *
FIXED_TYPED(PI));
```

Alternatively, but only if representation clauses are well-supported, the following rather clumsy code can be used in the present Ada language, to multiply or divide fixed values by named numbers, without having to specify any reduction in the named number's accuracy...

```
        PI: constant := 3.14.....;
        type PI_TYPE is delta PI range 0..2*PI
        for PI_TYPE'SMALL use PI;
        TYPED_PI: constant PI_TYPE := PI; -- Still as exact as        the named number.
begin
        FIXED_VALUE := FIXED_TYPE (FIXED_VALUE * TYPED_PI);
```

POSSIBLE SOLUTIONS:

Remove the restriction. Allow any of the following, where A and B are of possibly differing fixed-point types, and C is universal real literal or named number...

```
A := A_TYPE(C*B)
A := A_TYPE(B*C)
A := A_TYPE(B/C)
```

... but not A := A_TYPE(C/B); which is a different problem. (This is similar to the existing rules for mixed fixed and INTEGER operations; INTEGER/fixed is not available)>

To implement these operations, the compiler has simply to multiply or divide the scaling factor associated with the conversion of A to B, (obtained from the ratio of their 'BASE'SMALLs) by the value of C; then generate exactly the same code that it would have used for A := A_TYPE(B) but using the revised scaling factor (with the slight complication that the factor could now be negative).

Unlike some of the operations on fixed point numbers which involve the use of scaling factors, the one required here, namely a fixed-to-fixed conversion, is one of those which can be implemented easily.

At run-time it can be implemented using a multiply by one constant then a divide by another constant: where the constants are calculated by the compiler so that their ratio is a continued fraction approximation to the scaling factor. It can also be implemented by one multiply or divide, plus one shift, a test and two adds, as shown by Paul Hilfinger.

Both his and my methods only need the hardware arithmetic operations that are already need for other operations on the same types, and constants of those types; and yet the result is the same as if the literal or named number had been stored with infinite precision.

POSSIBLE SOLUTIONS:

INTEGER EXPONENTS

DATE: August 23, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 4.5.6(4)

PROBLEM:

The behavior of the exponentiation operator is always dependent on the target since the exponent must be of type standard.integer. It is generally a bad idea to use type standard.integer in programs, so it becomes necessary to overload exponentiation or use standard.integer.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use type standard.integer, either declaring variables of that type (forbidden by some style rules), or using an explicit conversion. Alternatively, overload "**" explicitly.

POSSIBLE SOLUTIONS:

Implicitly overload "**" for floating/integer type combinations. Of course, this tends to overload the symbol table.

ACCURACY REQUIRED OF COMPOSITE FIXED-POINT OPERATIONS

DATE: October 4, 1989

NAME: Terence J Froggatt

DISCLAIMER:

The views expressed in this note are those of the author, and do not necessarily represent those of SD-Scicon plc.

ADDRESS: Ada division, SD-Scicon plc,
Pembroke House, Pembroke Broadway,
Camberley, Surrey, GU15 3XD, U.K.

TELEPHONE: Home: +44 252 613996.
Work: +44 276 686200,

ANSI/MIL-STD-1815A REFERENCE: 4.5.7, primarily.

PROBLEM:

In formulating the accuracy requirements for real types, the Ada language designers appear to have overlooked the fact that certain operations on fixed-point types are composite, and so cannot be reasonably implemented to the required accuracy.

This should be contrasted with the situation for floating-point exponentiation: where the language designers recognized the problem and specified the accuracy in terms of a sequence of more primitive multiplications and divisions. 4.5.7(9).

For example, in the present Ada language it is possible to write SPEED := FLOAT (DIST/TIME); where SPEED is a floating-point type, and DIST & TIME are fixed. This is in reality 3 operations (not necessarily performed in this order):

1. A division of DIST by TIME,
2. A scaling operation using some mixture of multiplication, shifting, or division, by a factor which is a representable approximation to the universal real ratio of the 'BASE'SMALLs of DIST & TIME,
3. A conversion from fixed (i.e. integer) to float.

Even though there are several potential sources of inaccuracy here, the language requires the same overall accuracy here that it would require of any single step operation.

It is only because of the weakness of the ACVC tests in this area, and because support for arbitrary 'SMALLs is not compulsory, that the impracticability of this requirement is not more widely known.

Clearly the required accuracy could be obtained here and in all other difficult cases, by providing a universal real package in the target machine, providing arbitrary length rational arithmetic, (as was done for the NY interpreter). However, the language is (otherwise) designed to avoid this need.

I have shown that a finite arithmetic package will always suffice, but it needs a type capable of representing,

exactly, rather more than the product of any 3 fixed (or 2 fixed & 1 floating) types supported by the machine. The simulation of such a type by software would be very slow, and totally at odds with the objectives of fixed-point arithmetic.

DETAIL:

The operations which need to be considered, and my estimate of their difficulty (in the general case), are as follows:

a)	Fixed (Fixed);	Easy enough.
b)	Fixed (Fixed*Fixed);	Easy enough.
c)	Fixed (Fixed/Fixed);	Easy enough
d)	Fixed (Integer);	Easy enough.
e)	Integer (Fixed);	Very messy.
f)	Integer (Fixed*Fixed);	Impractical.
g)	Integer (Fixed/Fixed);	Impractical.
h)	Fixed (Float);	Unknown.
i)	Float (Fixed);	Unknown.
j)	Float (Fixed*Fixed);	Unknown.
k)	Float (Fixed/Fixed);	Unknown.

Even if the scalings are confined to powers-of-two, to implement some of these operations we will need to be able to hold the double-precision product of any two fixed types, and to be able to divide such a product by a single-precision type to get a back to single-precision answer. The implementor will probably choose what fixed-point base types to offer, according to the difficulty in providing such products for them.

Where the scalings are not confined to powers-of-two, but the scalings of the operands and results are in commensurate units, as is often the case, the scale factors cancel out in the compiler (giving simpler code than powers-of-two, where an occasional shift is needed): so these present no problems either. It is the incommensurate cases which are harder, where the scale factors do not cancel out to a power of two...

Paul Hilfinger has shown that (b) Fixed(Fixed*Fixed), and (c) Fixed(Fixed/Fixed), can be implemented, for any scales. Each uses one multiplication, one shift, and one division,
using no more than the double-precision product that we already need, plus a few relatively fast instructions.

Case (a) Fixed to Fixed, can be viewed as a degenerate (b) or (c), and case (d) Integer to Fixed, is the special case of (a): Fixed-delta-1 to Fixed. So there are no problems here.

Case (e) Fixed to Integer, is much harder: because rounding to the nearest integer is mandated. It might be just possible to do this using a divide routine doctored to round to the nearest, rather than towards zero, and using an extra addition to represent one more bit of the scaling constant than can usually be held.

I do not believe that there is a general implementation of (f) Integer(Fixed*Fixed), or (g) Integer(Fixed/Fixed), that gives the required accuracy for any scale factor, other than by simulating an arithmetic type not supported by hardware. (Paul Hilfinger has published some algorithms, but these only work for certain special cases).

To the best of my knowledge, no-one has even attempted the accuracy analysis of the remaining cases (h,i,j,k); which involve both floating-point and fixed-point with arbitrary scale factors. (There are quite a lot of different situations to consider).

Despite the "impracticals" and "unknowns" in the above table,

note that by far the most useful fixed-point operations, namely (a,b,c), are easy enough to implement for any scale factors.

IMPORTANCE: ESSENTIAL

The revised standard is unlikely to be accepted, if this revision request is not supported, simply because, on the evidence that I have seen, the present language cannot sensibly be implemented as specified.

The problem is with some of the less-important fixed-point operators, when used with arbitrary scales. But because the implementation of arbitrary scales is in itself optional, the effect of this problem is to remove arbitrary scales from the language completely.

In these days of floating-point coprocessors, most people who want to use fixed-point need it for some specific reason (such as communication with physical transducers) where arbitrary scales are important. So this revision request seeks to ensure that fixed-point arithmetic in Ada 9X justifies its existence.

CURRENT WORKAROUNDS:

In theory ...
The programmer can avoid operations (e,f,g) by declaring a fixed type with a delta of 1.0 (for each integer base type); and then converting the fixed expression via a fixed-delta-1 type to integer. The workaround for operations (h,i,j,k) involving float and arbitrary scaled fixed is similar: convert via a fixed type having a power-of-two scaling: but this is harder to express, since every such operation might need a different declaration.
Unfortunately, the compiler cannot tell, when processing a fixed point type 'SMALL length clause in one compilation unit, that the programmer is going to use the workarounds to ensure that only the implementable operations are called. The compiler generally has to assume that the dubious operations are going to be used (in some other compilation unit). The language does not permit a compiler to reject the dubious operations, but it does allow the outright rejection of the length clause.

So many Ada compilers do not implement arbitrary scales at all: even though the most important operations are easy. Legal but not helpful. Other Ada compilers may try to be helpful by providing arbitrary scales, by disregarding their accuracy, exploiting the weaknesses of the ACVC. The problem here is that programmers will have no idea what accuracy they can count on, possibly even for the easy operators.

In practice therefore ...
The workaround most likely to be used by the programmer who wants arbitrary scalings, (either to communicate with physical transducers, or to obtain the cleaner abstraction offered by range-related scalings), will be to declare types like FRACTION and LONG_FRACTION, all with ranges of -1.0..+1.0, and do the scaling by explicit multiplications.

POSSIBLE SOLUTIONS:

If we were designing a low-level language from scratch, I would
propose that it offered just pure fractions (as used in the above workaround), where the programmer does the scaling. If we were designing a high-level language from scratch, I would propose arbitrary-scaled fixed-point where the compiler does all the scaling, as (supposedly) available as an option in the present Ada language (except that, personally, I would not make it optional, and I would use range-related scales by default).
I have never seen the advantage in offering just power-of-two scalings: from the compiler writer's viewpoint

they offer little intellectual simplification relative to arbitrary scales (given that compilers already optimize multiplications into shifts when they can, and given sensible accuracy requirements), while from the programmer's viewpoint they offer little abstractional advance relative to fractions (such as peripheral-related or range-related scales).

Given the existing Ada language as a starting point, we cannot just drop fixed-point altogether or offer just pure fractions; but we do have a choice: we can either go backwards by removing arbitrary scales from Ada 9X, offering just the power-of-two scales guaranteed by the existing language, or we can go forwards by making sure that arbitrary scales are implementable. Since the most important operations can already be made to work to the required accuracy, I believe we should go forwards, and all that we need for Ada 9X is a well-defined relaxation of the requirements on the troublesome rarer fixed-point operations.
I am not seeking a relaxation to make the compiler-writer's life easier: indeed, the exact opposite: I would like to make sure there is no excuse for not implementing arbitrary scale factors. I would far rather see the compiler doing the universal real scale factor arithmetic than ask the programmer to do it.
Setting an unrealistic accuracy target is no help to implementors or to users: to be able to write portable numeric software, we need a well-defined but reachable target for Ada 9X, backed up by ACVC tests. (The SigAda numerics WG has a similar realistic approach to the accuracy required of its elementary function package.)

It seems to me that a sensible goal would be to ensure that all of the operations on a fixed-point type with arbitrary scale can be performed by the same set of hardware primitives that are assumed for a fixed-point type of the same length but with a power-of-two scaling; (such as the availability of a double-precision product). There are several ways that this might be achieved ...

1) If it could be demonstrated that (e) and (h,i) were in fact easy, change 4.5.5(11) from "universal-fixed .. must always be explicitly converted to some NUMERIC type" to read ".. to some FIXED type", which would eliminate (f,g,j,k). I am sure that most Ada users are unaware that the results of fixed multiply or divide can be converted directly to integer or float.

2) If it could be demonstrated that (h,i,j,k) are all in fact easy, reword 4.6(7) to only require "nearest" for float-to-integer; and allow fixed-to-integer to round up or down, (like the half-way cases already do), thereby simplifying (e,f,g).

3) Assuming that only (a,b,c,d) are easy, we could relax (e,f,g) as in (2), and find some form of words to relax (h,i,j,k). This could be tricky to formulate in terms of intervals or in terms of equivalent operations; but the aim should be to allow the operator, the scaling, and the conversion, to be implemented as separate steps (in the order which gives the best accuracy for the types in question).

4) An alternative approach is to prohibit the difficult operations, possibly all of (e,f,g,h,i,j,k), but only in those instances where at least one of the fixed types concerned actually has a non-power-of-two scale. Or, easier but more restrictive: if it has an explicit 'SMALL length clause, whatever the value. (Although 13.1(10) says that a representation clause cannot change the net effect of the program, this statement is already known to be false for 'SMALL, which clearly changes the model numbers of a type.)

Much as I dislike affording a special significance to power-of-two scales, I suspect that (4) would be the right choice, starting from the existing Ada language, (even if it does lead to some complexity in checking generic bodies against instantiations), because:

*) The operations which remain for arbitrary scales, (a,b,c,d), include the most important ones, (a,b,c), without which it would be hard to do anything. It is fortunate that these can reasonably be

implemented to the current accuracy requirements.

*) Unlike proposals (1,2,3), no relaxation in the availability or accuracy of the operators is being proposed for power-of-two types. So the only programs affected will be those using arbitrary scales, (and making rather optimistic assumptions about their accuracy).

*) The existing Ada accuracy model survives. The language restriction is straightforward to explain, to implement, and to understand. It can be defined now, without further research by myself or by Paul Hilfinger, into what can and cannot be done.

The ideal solution, of course, would be for someone to prove me wrong, by publishing efficient algorithms for all of the operations.

ACCURACY REQUIRED OF COMPOSITE FIXED-POINT OPERATIONS

DATE: October 4, 1989

NAME: Terence J. Froggatt

DISCLAIMER:

The views expressed in this note are those of the author and do not necessarily represent those of SD-Scion Plc.

ADDRESS: Ada Division, SD-Scion Plc.
Pembroke House, Pembroke Broadway,
Camberley, Surrey, GU15 3XD, U.K.

TELEPHONE: Home: +44 252 613996
Work: +44 276 686200

ANSI/MIL-STD-1815A REFERENCE: 4.5.7 (primarily)

PROBLEM:

In formulating the accuracy requirements for real types, the Ada language designers appear to have overlooked the fact that certain operations on fixed-point types are composite, and so cannot be reasonably implemented to the required accuracy.

This should be contrasted with the situation for floating-point exponentiation: where the language designers recognized the problem and specified the accuracy in terms of a sequence of more primitive multiplications and divisions. 4.5.7(9).

For example, in the present Ada language it is possible to write SPEED := FLOAT (DIST/TIME); where SPEED is a floating-point type, and DIST & TIME are fixed. This is in reality 3 operations (not necessarily performed in this order):

1. A division of DIST by TIME,
2. A scaling operation using some mixture of multiplication, shifting, or division, by a factor which is a representable approximation to the universal real ratio of the BASE'SMALLs of DIST & TIME,
3. A conversion from fixed (i.e. integer) to float.

Even though there are several potential sources of inaccuracy here, the language requires the same overall accuracy here that it would require of any single step operation.

It is only because of the weakness of the ACVC tests in this area, and because support for arbitrary 'SMALLs is not compulsory, that the impracticality of this requirement is not more widely known.

Clearly the required accuracy could be obtained here and in all other difficult cases, by providing a universal real package in the target machine, providing arbitrary length rational arithmetic, (as was done for the NY interpreter). However, the language is (otherwise) designed to avoid this need.

I have shown that a finite arithmetic package will always suffice, but it needs a type capable of representing,

exactly, rather more than the product of any 3 fixed (or 2 fixed and 1 floating) types supported by the machine. The simulation of such a type by software would be very slow, and totally at odds with the objectives of fixed-point arithmetic.

DETAIL:

The operations which need to be considered, and my estimate of their difficulty (in the general case), are as follows:

a)	Fixed (Fixed);	Easy Enough.
b)	Fixed (Fixed*Fixed);	Easy Enough.
c)	Fixed (Fixed/Fixed);	Easy Enough.
d)	Fixed (Integer);	Easy Enough.
e)	Integer (Fixed);	Very Messy.
f)	Integer (Fixed*Fixed);	Impractical.
g)	Integer (Fixed/Fixed);	Impractical.
h)	Fixed (Float);	Unknown.
i)	Float (Fixed);	Unknown.
j)	Float (Fixed*Fixed);	Unknown.
k)	Float (Fixed/Fixed);	Unknown.

Even if the scalings are confined to powers-of-two, to implement some of these operations we will need to be able to hold the double-precision product of any two fixed types, and to be able to divide such a product by a single-precision type to get back to single-precision answer. The implementor will probably choose what fixed-point base types to offer, according to the difficulty in providing such products for them.

Where the scalings are not confined to powers-of-two, but the scalings of the operands and results are in commensurate units, as is often the case, the scale factors cancel out in the compiler (giving simpler code than powers-of-two, where an occasional shift is needed): so these present no problems either. It is the incommensurate cases which are harder, where the scale factor do not cancel out to a power of two...

Paul Hilfinger has shown that (b) Fixed (Fixed*Fixed), and (c) Fixed (Fixed/Fixed), can be implemented, for any scales. Each uses one multiplication, one shift, and one division, using no more than the double precision product that we already need, plus a few relatively fast instructions.

Case (a) Fixed to Fixed, can be viewed as a degenerate (b) or (c), and case (d) Integer to Fixed, is the special case of (a): Fixed-delta-1 to Fixed. So there are no problems here.

Case (e) Fixed to Integer, is much harder: because rounding to the nearest integer is mandated. It might be just possible to do this using a divide routine doctored to round to the nearest, rather than towards zero, and using an extra addition to represent one more bit of the scaling constant than can usually be held.

I do not believe that there is a general implementation of (f) Integer (fixed*Fixed), or (g) Integer (Fixed/Fixed), that gives the required accuracy for any scale factor, other than by stimulating an arithmetic type not supported by hardware. (Paul Hilfinger has published some algorithms, but these only work for certain special cases).

To the best of my knowledge, no-one has even attempted the accuracy analysis of the remaining cases (h,i,j,k); which involve both floating point and fixed-point with arbitrary scale factors. (There are quite a lot of different situations to consider).

Despite the "impracticals" and "unknowns" in the above table, note that by far the most useful fixed-point

operations, namely (a,b,c), are easy enough to implement for any scale factors.

IMPORTANCE: ESSENTIAL

The revised standard is unlikely to be accepted, if this revision request is not supported, simply because, on the evidence that I have seen, the present language cannot sensibly be implemented as specified.

The problem is with some of the less-important fixed-point operators, when used with arbitrary scales from the language completely.

In these days of floating-point coprocessors, most people who want to use fixed-point need it for some specific reason (such as communication with physical transducers) where arbitrary scales are important. So this revision request seeks to ensure that fixed-point arithmetic in Ada 9X justifies its existence.

CURRENT WORKAROUNDS:

In theory...
The programmer can avoid operations (e,f,g) by declaring a fixed type with a delta of 1.0 (for each integer base type); and then converting the fixed expression via a fixed-delta-1 type to integer. The workaround for operations (h,i,j,k) involving float and arbitrary scaled fixed is similar: convert via a fixed type having a power-of-two scaling: but this is harder to express, since every such operation might need a different declaration.

Unfortunately, the compiler cannot tell, when processing a fixed point type 'SMALL length clause in one compilation unit, that the programmer is going to use the workarounds to ensure that only the implementable operations are called. The compiler generally has to assume that the dubious operations are going to be used (in some other compilation unit). The language does not permit a compiler to reject the dubious operations but it does allow the outright rejection of the length clause.

So many Ada compilers do not implement arbitrary scales at all: even though the most important operations are easy. Legal but not helpful. Other Ada compilers may try to be helpful by providing arbitrary scales, by disregarding their accuracy, exploiting the weaknesses of the ACVC. The problem here is that programmers will have no idea what accuracy they can count on, possibly even for the easy operators.

In practice therefore...
The workaround most likely to be used by the programmer who wants arbitrary scalings, (either to communicate with physical transducers, or to obtain the cleaner abstraction offered by range-related scalings), will be to declare types like FRACTION and LONG_FRACTION, all with ranges of -1.0..+1.0, and do the scaling by explicit multiplications.

POSSIBLE SOLUTIONS:

If we were designing a low-level language from scratch, I would propose that it offered just pure fraction (as used in the above workaround), where the programmer does the scaling. If we were designing a high-level language from scratch, I would propose arbitrary-scaled fixed point where the compiler does all the scaling, as (supposedly) available as an option in the present Ada language (except that, personally, I would not make it optional, and I would use range-related scales by default).

I have never seen the advantage in offering just power-of-two scalings: from the compiler writer's viewpoint they offer little intellectual simplification relative to arbitrary scales (given that compilers already optimize multiplications into shifts when they can, and given the sensible accuracy requirements), while from the

programmer's viewpoint they offer little abstractional advance relative to fractions (such as peripheral-related or range-related scales).

Given the existing Ada language as a starting point, we cannot just drop fixed-point altogether or offer just pure fraction; but we do have a choice; we can either go backwards by removing arbitrary scales from Ada 9X, offering just the power-of-two scales guaranteed by the existing language, or we can go forwards by making sure that arbitrary scales are implementable. Since the most important operations can already be made to work to the required accuracy, I believe we should go forwards, and all that we need for Ada 9X is a well-defined relaxation of the requirements on the troublesome rarer fixed-point operations.

I am not seeking a relaxation to make the compiler-writer's life easier: indeed, the exact opposite: I would like to make sure there is no excuse for not implementing arbitrary scale factors. I would far rather see the compiler doing the universal real scale factor arithmetic than ask the programmer to do it.

Setting an unrealistic accuracy target is no help to implementors or to users: to be able to write portable numeric software, we need a well-defined but reachable target for Ada 9x, backed up by ACVC tests. (The SigAda numerics WG has a similar realistic approach to the accuracy required of its elementary function package.)

It seems to me that a sensible goal would be to ensure that all of the operations on a fixed-point type with arbitrary scale can be performed by the same set of hardware primitives that are assumed for a fixed-point type of the same length but with a power-of-two scaling; (such as the availability of a double-precision product). There are several ways that this might be achieved...

1) If it could be demonstrated that (e) and (h,i) were in fact easy, change 4.5.5(11) from "universal-fixed .. must always be explicitly converted to some NUMERIC type" to read ".. to some FIXED type", which would eliminate (f,g,j,k). I am sure that most Ada users are unaware that the results of fixed multiply or divide can be converted directly to integer or float.

2) If it could be demonstrated that (h,i,j,k) are all in fact easy, reword 4.6(7) to only require "nearest" for float-to-integer; and allow fixed-to-integer to round up or down, (like the half-way cases already do), thereby simplifying (e,f,g).

3) Assuming that only (a,b,c,d) are easy, we could relax (e,f,g) as in (2), and find some form of words to relax (h,i,j,k). This could be tricky to formulate in terms of intervals or in terms of equivalent operations; but the aim should be to allow the operator, the scaling, and the conversion, to be implemented as separate steps (in the order which gives the best accuracy for the types in question).

4) An alternative approach is to prohibit the difficult operations, possibly all of (e,f,g,h,i,j,k), but only in those instances where at least one of the fixed types concerned actually has a non-power-of-two-scale. Or, easier but more restrictive: if it has an explicit 'SMALL length clause, whatever the value. (Although 13.1(10) says that a representation clause cannot change the net effect of the program, this statement is already known to be false for 'SMALL, which clearly changes the model numbers of a type.)

Much as I dislike affording a special significance to power-of-two scales, I suspect that (4) would be the right choice, starting from the existing Ada language, (even if it does lead to some complexity in checking generic bodies against instantiations), because:

*) The operations which remain for arbitrary scales, (a,b,c,d), include the most important ones, (a,b,c), without which it would be hard to do anything. It is fortunate that these can reasonably be

implemented to the current accuracy requirements.

*) Unlike proposals (1,2,3), no relaxation in the availability or accuracy of the operators is being proposed for power-of-two types. So the only programs affected will be those using arbitrary scales, (and making rather optimistic assumptions about their accuracy).

*) The existing Ada accuracy model survives. The language restriction is straightforward to explain, to implement, and to understand. It can be defined now, without further research by myself or by Paul Hilfinger, into what can and cannot be done.

The ideal solution, of course, would be for someone to prove me wrong, by publishing efficient algorithms for all of the operations.

ROUNDING OF NUMERIC CONVERSIONS

DATE: August 7, 1989

NAME: J G P Barnes

ADDRESS: Alsys Ltd, Newtown Road
Henley-on-Thames, Oxon,
RG(IEN, UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 4.6

PROBLEM:

The language explicitly states that when converting to an integer type, rounding of real values halfway between two integral values may be either up or down. This lack of predictability causes unnecessary portability problems.

IMPORTANCE: IMPORTANT

It is important to make the language as predictable as possible. There is no obvious reason for the unnecessary vagueness in this area.

CURRENT WORKAROUNDS:

Private conversion function have to be written if a particular rule (such as rounding up is required. These cannot replace the predefined conversion since the type name used as the conversion "operator" cannot be overloaded.

Such routines are inevitably slow and have to be provided for each real type concerned.

POSSIBLE SOLUTIONS:

The language should prescribe specific behavior on midway values. There are a number of possible choices (which may be why the designers left it undefined):

- rounding up: perhaps the most expected choice,
- rounding down: could be argued for,
- nearest even: unexpected perhaps.

Rounding to nearest even means that 0.5 goes to 0, 1.5 and 2.5 go to 2, 3.5 and 4.5 go to 4 and so on. This has some appeal since it does not bias the result one way or the other on average.

Anyone of these can be easily programmed in terms of one of the others. Thus the choice is not critical but one should be predefined (and the same for all implementation!). Existing nonerroneous programs will

not be affected.

SETTING NEW LOWER BOUNDS FOR ARRAY TYPES

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 4.6

PROBLEM:

Numerical linear algebra almost always manipulates unconstrained arrays; these arrays frequently have different (lower) bounds especially when BLAS (Basic Linear Algebra Subprograms in FORTRAN) are utilized in Ada. It is not possible in the current language to implement the BLAS and other similar vector and matrix operations in a natural and efficient way in Ada. Consider the following simple example:

```
type VECTOR_TYPE is array (INTEGER range <>) of FLOAT;

function "+" (V, W : VECTOR_TYPE) return VECTOR_TYPE is
        subtype V_TYPE is VECTOR_TYPE (V'RANGE);
        WW : V_TYPE renames W;
                -- this has no effect, constraints on V_TYPE must be ignored
        RESULT : V_TYPE;
begin
        for I in V'RANGE loop
                RESULT(I) := V(I) +
                -- W(I) no : the ranges are different
                -- V_TYPE(W)(I) illegal : conversion of prefix not allowed
                -- WW(I) no : the rename has no effect

                W(I + W'FIRST - V'FIRST);
                -- which is much less clear and on many
                -- compilers less efficient.
```

We require the ability to reset the lower bound(s) of any object or parameter of a constrained array type to the value of a given expression of an appropriate discrete (sub)type.

IMPORTANCE: IMPORTANT

The reason for the widespread adoption of the BLAS and analogous array operations in FORTRAN is that they can be implemented efficiently and it is their efficiency which is critical to the overall efficiency of large numerical linear algebra applications. We need to be able to implement such operations in a portable and efficient way in Ada.

CURRENT WORKAROUNDS:

A current workaround is shown above, but its impact on efficiency is obvious especially when such loops are nested.

POSSIBLE SOLUTIONS:

Allow renaming to have an effect on constraints or introduce a new construct which has such an effect (for arrays).

INTEGERCONV

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 4.6(6-7

PROBLEM:

Ada supplies only a rounding conversion from a real type to an integer type. Ada does not supply a floor, ceiling, truncate, or whole operations from real to integer.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

User conversion routines.

```
N := integer( R ); S := float( N )
if S > R then N := N - 1; end if;
```

would suffice to floor a real number, but it involves a double conversion. Inefficient.

POSSIBLE SOLUTIONS:

Definitions (assume R is any real number and N is any integer number):

```
N = floor   ( R ) -- N is the largest integer <= R
N = ceiling ( R ) -- N is the smallest integer >= R
N = truncate( R ) -- if R >= O then N = floor( R ) else N = ceiling( R );
N = whole   ( R ) -- if R >= O then N = ceiling( R ) else N = floor( R );
```

Ada should provide a way to have more control over conversions from any real type to any integer type. There are two possibilities for a solution:

Solution 1. -- not recommended

Add to package STANDARD:

```
type Integer_Conversion is ( Round, Floor, Ceiling, truncate, Whole );
```

Then for every implicitly declared real to integer conversion routine which is in fact the name of an integer type, allow a second parameter of type INteger_conversion.

Examples: N = integer(R, floor);

```
N = integer( R, truncate );
N = integer( R ); -- Round is the default, of course.
```

Solution 2. -- recommended

Add four attributes to the language:

P'Floor
P'Ceiling
P'Truncate
P'Whole

For each of the above, define the attributes for any prefix P that denotes an integer type, and make the attribute a function which takes any real (floor or fixed) type and converts it to an integer with the appropriate conversion.

ROUNDING DUE TO REAL-TO-INTEGER CONVERSION

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 4.6(7)

PROBLEM:

A given implementation is not required to state whether rounding is up or down if "...the operand is halfway between two integers..."

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Compiler specific trial-and-error type tests will yield the information but this does not allow for any portability.

POSSIBLE SOLUTIONS:

1. Preferred: include an attribute "Rounds-Up" which acts on a real subtype and returns either true or false for this "halfway" condition.

2. Not-so-good: require an Appendix F entry stating the direction of rounding.

THE RUN TIME SYSTEM (RTS) IS COMPLETELY PROVIDED BY THE COMPILER VENDOR

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 4.8, 9

PROBLEM:

Currently, the run time system (RTS) is completely provided by the compiler vendor. Hence there is a variety of completely different run time systems, with different real time behavior and with no incentive for suppliers of target test tools (e.g., non-intrusive program analyzers) to invest in Ada support facilities. Moving to a faster processor supported by a different compiler may actually reduce performance. Desirable to standardize on an interface between the compiler-specific and target-specific parts of the RTS, thus allowing the latter to be written by appropriate specialist suppliers. This would improve portability of real time software between compilers, and encourage investment in test tools.

IMPORTANCE: ESSENTIAL

High testing costs, poor portability, and risk of Ada being avoided for embedded systems.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Need to define semantics of the interface fully, so that the target specific part (kernel) of the RTS can be validated alone, and compilers validated using any appropriate kernel.

THE RUN TIME SYSTEM (RTS) VARIES CONSIDERABLY FROM ONE COMPUTER VENDOR TO ANOTHER

DATE: August 1, 1989

NAME: Mr. J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 4.8, 9

PROBLEM:

The run time system (RTS) varies considerably from one compiler vendor to another, particularly in the areas of tasking and heap management. Performance in these areas, in terms of both CPU and memory usage, needs to be documented in order for a system designer to decide how best to use Ada for a given application with a given compiler. Currently, such documentation is sometimes missing altogether, and is often inadequate.

IMPORTANCE: ESSENTIAL

High risks for performance critical Ada projects, and risk of Ada being avoided for embedded systems.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Ada compiler vendors should be required to provide, in addition to appendix F, adequate documentation of the behavior of the run time system to allow the run time performance and memory utilization of an Ada program to be predicted from its design. This could take the form of an Appendix G.

UNCONSTRAINED TYPES WITH DISCRIMINANTS AS GENERIC PARAMETERS IN RELATION WITH ACCESS TYPES.

DATE: October 24, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail : madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 4.8(5), AI-00037

PROBLEM:

Consider the two following variants of a simple generic package:

```
generic
        type T is private;
package Variant_A is
        procedure Store (X : in T);
        procedure Retrieve (X : out T);
end Variant_A;

package body Variant_A is

        Storage : T;

        procedure Store (X : in T) is
        begin
                Storage := X;
        end Store;

        procedure Retrieve (X : out T) is
        begin
                X := Storage;
        end Retrieve;

end Variant_A;

generic
        type T is private;
package Variant_B is
        procedure Store (X : in T);
        procedure Retrieve (X : out T);
 end Variant_B;
```

```
package body Variant_B is

        type Access_T is access T;

        Storage : Access_T := new T;

        procedure Store (X : in T) is
        begin
                Storage.all := X;       -- *
        end Store;

        procedure Retrieve (X : out T) is
        begin
                X := Storage.all;
end Retrieve;

end Variant_B;
```

These two variants look equivalent at first sight, but they are not: if Variant_B is instantiated with an actual type with discriminants that have defaults, then Constraint_Error will be raised at the *-marked assignment if X does not have the same discriminant values as the type's default. This is not true for Variant_A. This seems strange because AI-00037 suggests that a type with discriminants that have defaults may be treated exactly as a constrained type, and also because in this special case there is a difference between declared objects and objects created by an allocator.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Do not use access types designating generic formal types, or do not instantiate generics with unconstrained types with discriminants that have defaults.

POSSIBLE SOLUTIONS:

Change 4.8(5) so that objects of a type with discriminants that have defaults are unconstrained, regardless of whether they are created by allocators or not.

SCRUBBING MEMORY TO IMPROVE TRUSTWORTHINESS OF TRUSTED COMPUTING BASE SYSTEMS

DATE: May 18, 1989

NAME: George A. Buchanan

ADDRESS: IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

TELEPHONE: (301) 731-8894 ext. 2063

ANSI/MIL-STD-1815A REFERENCE: 4.8(7), 8, 9.7.1, 9.10

PROBLEM:

When memory used by a trusted computing base (TCB) is released back to the heap during program execution, sensitive data may remain in the memory and be vulnerable to access by unauthorized users. This released memory may be either deallocated dynamic storage or no longer visible memory that contains data objects of a package, subprogram, or task. These data objects lose their visibility when program execution leaves the scope of the package or subprogram (i.e., returns the storage used by an activation record back into the heap), or when a task is terminated or aborted. No mechanism is currently provided to scrub this memory. Also, the state and contents of the heap are not otherwise monitored by the reference monitor (security kernel) because the heap is not recognized as an object managed by the TCB.

Depending on the way that memory is used, it can be difficult for an Ada programmer to recognize when the memory is going to be returned to the heap. Ada currently provides automatic garbage collection of dynamic storage. The Ada language does not require this memory to be scrubbed when it is deallocated. It can be difficult for a programmer to recognize when allocated memory is no longer "pointed to" and is collected as garbage. It is also difficult for a programmer to recognize when program execution is going to leave the scope of a package. At this time, sensitive data objects declared within the package (outside of subprograms declared within the package) should be scrubbed.

IMPORTANCE: IMPORTANT

No TCB's sensitive data should be left in the heap because it potentially may be accessed by unauthorized users. A programmer who is creating or maintaining a TCB should be able to specify data that is to be scrubbed from memory before the memory is returned to the heap. This capability will ensure that discarded but sensitive data will not be available to be accessed by unauthorized users.

CURRENT WORKAROUNDS:

Workarounds are currently available for only some of the cases discussed above. The sensitive data objects declared within subprograms can be easily recognized by a programmer and should be scrubbed just before returning from the subprogram calls. Also, the sensitive data objects declared within tasks can be easily recognized by a programmer and should be scrubbed before terminating or aborting tasks. For the programmer to enforce the scrubbing of memory before it is returned to the heap using currently available Ada constructs and features is awkward and unreliable. This is particularly true for dynamic storage when

it is deallocated and for data objects declared within a package (but outside of subprograms within the package) when the data objects are no longer visible (i.e., when program execution leaves the scope of the package). Also, allowing the programmer to define his own scrub values presents the possibility of creating a covert channel with the heap.

POSSIBLE SOLUTIONS:

One solution to this problem would be to require the security kernel (hardware/software) to scrub all memory before it is returned to the heap. The drawback of this approach is that it could impose excessive overhead because even memory containing nonsensitive data would be scrubbed.

A more flexible and time-efficient solution would be to introduce a pragma that the programmer could use to specify the data that should be scrubbed and the data structures, compilation units, and/or tasks from which the specified data should be scrubbed. The pragma could have the following forms.

```
pragma SCRUB ( <data object> {, <data object> } );

--      This form scrubs the indicated data object(s) when program execution
--      leaves the scope of the compilation unit or task that contains the
--      declaration of the data object(s). The data object is scrubbed by
--      being assigned the predefined scrubbed value (e.g., all bits of the
--      corresponding memory are filled with 0s). The pragma is only allowed
--      immediately within the declarative part of the subprogram or within
--      the package specification, package body, or task body that contains the
--      data object(s); the last object's declaration must occur before the
--      pragma.

   pragma SCRUB ( all );

--      This form scrubs all data objects declared within the compilation unit
--      or task, that contains their declarations, when program execution leaves
--      the scope of the compilation unit or task. The pragma would be placed
--      in the compilation unit's or task declaration's section. The "all"
--      indicates that all data objects are to be assigned the predefined scrub
--      value, e.g., all bits of the corresponding memory are filled with 0s.
--      For example,

   package Sensitive_Data_Manager_Package is
     pragma SCRUB ( all );

     . . .
```

The compiler or runtime environment must be able to recognize when memory is deallocated or returned to the heap when program execution has left the scope of a compilation unit and when a task is terminated or aborted. Before a node is deallocated the runtime environment must traverse from this node to scrub and deallocate any remaining nodes that are not otherwise "pointed to." The execution of the pragma SCRUB, like all other pragmas, should be validated.

Another flawed, but at least an interim, solution would be to use a preprocessor to define a new task call for each instance of pragma SCRUB that will scrub memory upon leaving the scope of a unit or when a task terminates or aborts. This introduces additional execution overhead due to the new task. Also the memory used by this new task needs to be scrubbed when the task terminates or aborts, thus posing the

problem of this new task continuing to call itself.

Clearly the best solution is to require the inclusion of pragma SCRUB, as described above, in the Ada language revision.

STATICNESS IN GENERIC UNITS

DATE: October 11, 1989

NAME: J G P Barnes (endorsed by Ada (UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 4.9, 12.1

PROBLEM:

In order to encourage the use of the generic mechanism for writing reusable components it is important that functionality is not unnecessarily restricted in generic units. In other words it ought to be possible, in general, to take a nongeneric unit and make it generic.

An important restriction that is often an impediment is that generic parameters are never static.

This prevents the use of a case statement where the type of the expression is a formal generic type. Although it is clear that this is not usually a hardship in the case of enumeration types (because the required lieterals would have to be passed as parameters as well), nevertheless it is perfectly useful with integer types where the literals are of course universal.

Similarly, the type of a discriminant cannot be generic and again this seems to an unnecessary restriction.

The attributes of a formal type are also not static; there are situations in numerical algorithms where the use of attributes such as MANTISSA or DIGITS is thereby impeded.

IMPORTANCE: IMPORTANT

At the present the full use of generics is impeded by these restrictions.

CURRENT WORKAROUNDS:

Case statements can always be expressed as a series of if statements but this obscures the abstraction. Other problems can often only be overcome by not making the unit generic and thus increasing the maintenance problem.

POSSIBLE SOLUTIONS:

The restrictions on staticness could be replaced by restrictions on the actual parameter rather than the formal parameter. That is the formal parameter would be considered static if the actual parameter were. In order to maintain the contract model, this would require a notation for the formal parameter which expressed that the actual had to be static. In the case of an in parameter this could perhaps be

```
generic
        X: constant T;
```

The restriction on case statements and discriminants could be replaced by a similar requirement that the actual subtype be static which could perhaps be expressed by

```
Type T is constant range <>;
```

which would require the actual parameter to be a static integer subtype. Similar notational extension would apply to other generic parameter declarations.

This change would require a reappraisal of the philosophy of compilation and code sharing in generics. The current restrictions enable more complete compilation before instantiation. It might be that the users would be better served by a more straightforward approach which actually gave them the functionality they required at the linguistic and design level. The value of codesharing is probably overrated especially since practical implementations have not really achieved this.

Any difficulty with the inability to share code in traditional situations just involving subprogram parameters would be better solved by introducing subprograms as parameters to subprograms (as requested separately by many users) so that the issue of codesharing is less important.

LARGE AND/OR COMPLEX CONSTANTS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 4.9, 4.10

SUMMARY:

Many embedded systems require extensive "constants" such as precomputed tables, initialized data structures, etc., which are relatively easy to compute at run-time, but difficult to compute at compile time. Since these "constants" may want to live in Read-Only Memory, computing them at run-time is not really a possibility. However, including page after page of hexadecimal constants in the source code does not provide a maintenance programmer with much help in the understanding of the program.

In order to support the requirements for maintaining systems involving such constants, Ada should be able to do much more in the way of evaluation during compile time, including interpreting functions used to initialize constant arrays.

This request is not really a proposal to change the Ada language itself, but its compiling philosophy.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Some mechanisms should be provided to allow a system designer to specify complex tables and data structures to be built and loaded that will appear as constants during the actual running of the program. This problem has not typically been a problem in more traditional data processing where tables can be computed during an initialization phase, but in many embedded systems, constants do not reside in writable memory, and therefore must be precomputed.

The size and complexity of some of these constants can be very large, as modern Read-Only Memory and optical disk technology comes on-stream. Therefore, such constants may not be simple arrays or record structures, but might be complex data structures involving access types and perhaps even "pre-elaborated" tasks. An extreme example might be a grammar/dictionary for the English language, which could consist of 500 megabytes of very complex data structure.

Some mechanisms must be found to develop and integrate the programs that build the data structures with the programs that incorporate them.

CURRENT WORKAROUNDS:

The only current workarounds for this problem are the building of separate programs to precompute tables and structures and produce constant "aggregate" source code which is then inserted into an Ada program

by means of a text editor. This is a messy procedure which violates all of the careful abstraction mechanisms built into the Ada language.

Steelman requirement 11C encourages as much computation at compile time as possible.

NON-SUPPORT IMPACT: Continuing messy programs.

POSSIBLE SOLUTIONS:

Compilers which can interpret and execute functions and procedures all of whose arguments are known at compile time. Load modules which can handle complex objects such as access objects and task objects.

DIFFICULTIES TO BE CONSIDERED: MANY

REMOVE RESTRICTIONS ON DESIGNATING ADA EXPRESSIONS "STATIC"

DATE: October 28, 1989

NAME: Michael F. Brenner

DISCLAIMER:

The opinions are mine; not necessarily InterACT's

ADDRESS: InterACT Corporation
417 Fifth Avenue
New York, New York 10016

TELEPHONE: (212) 696-3680

ANSI/MIL-STD-1815A REFERENCE: 4.9

Also, other parts of the standard as referenced by those Approved Ada Commentaries referenced below, especially AI-00128 and AI-190.

PROBLEM:

Ada's acceptance depends on one factor more than any other, its ability to generate fast code. It was originally intended for Ada to be efficient "pragma suppress", there should be no theoretical reason for Ada to generate less efficient machine code than other languages. However, there are some restrictions which force implementers to generate worse code than users desire.

Since compile time evaluation of expressions is an "easy" optimization, and one of the most effective at reducing executing time, it is important that the language allow as many expressions as possible to be designated "static", particularly after enclosing the code in a generic definition. Other optimizations are also important of course, and should be allowed.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

The current workaround is for slow code to be generated.

POSSIBLE SOLUTIONS:

Trigonometric functions (or any function declared with pragma no_side_effects), numerical type conversions, attributes of static expressions (esp image, value, first, last, pred, etc.), or arrays, records, slices, concatenations or boolean operations of static expressions should be static.

Code should never run slower because it is generic. To do this, some checking must be done at instantiation time, to allow expressions to be static which would have been static if they were outside the generic definition.

The Ada 9X Committee should review the published Ada Commentaries as to which interpretations and ramifications would result in faster execution, if the appropriate changes to the Standard were made.

As a first approximation to that review, I have listed them here as "SLOW" if they should be changed for speed reasons, and "FAST" if they should not be changed. Most of the commentaries affect compilation only or are execution speed neutral ("NEUT").

AI-OOOO1	FAST	Makes more expressions static. COULD BE FASTER: by changing the ALRM to state that any value which is known at compile time is static
AI-00002	NEUT	syntax only
AI-00006	FAST	Gives static loop parameters static subtypes.
AI-00007	FAST	Defers some checking; makes other checking optional. COULD BE FASTER: by emphasizing that in most cases the checking can (and therefore should) be done at compile time WOULD HAVE BEEN FASTER: if option 1 were accepted, but would have lost functionality WOULD HAVE BEEN SLOWER: if option 3 were accepted
AI-00008	NEUT	syntax only
AI-00012	NEUT	syntax only
AI-00014	NEUT	syntax only (probably no compiler did the extra evaluations)
AI-00015	NEUT	note that this is an example of "helping" the compiler manufacturers, in a way that turned out not to "help" them at all. The overload resolutions could allow resolution to use the syntactic context of expressions without slowing compile speed noticeably. That would make Ada easier to read and write, but would have no effect on execution speed.
AI-00016	FAST	Any flexibility in the "renames" clauses will cause programmers to use them more and thus will further a style of programming that is easier to optimize than most other styles.
AI-00018	FAST	This gives flexibility to compiler writers with no reduction in functionality. COULD BE FASTER: by emphasizing that in most cases the checking can (and therefore should) be done at compile time
AI-00019	SLOW	The interpretation should have said that CONSTRAINT_ERROR should be noted at compile time. There is no excuse for allowing this program to link or execute.
AI-00020	SLOW	The response to this commentary does not allow fast, accurate implementations of fixed point arithmetic at compile time, in static expressions, or at run time.
AI-00023	SLOW	Although the interpretation speeded up certain programs by declaring that certain expressions are static, it held to the theory that type conversions of static expressions are not static. The ALRM should be modified to demand that all type conversions of static expressions, where the full type is known at compile time, are static expressions.
AI-00024	NEUT	The interpretation is lengthy and precludes future expansions of Ada in the direction it should evolve.
AI-00025	SLOW	But necessary
AI-00026	SLOW	An unnecessary restriction. It demotivates programmer use of compile time attributes and therefore fosters styles of coding which are less optimizable. If the compiler knows the value of an attribute, it should be considered static.
AI-00027	NEUT	Allows a future invention of an optimization

AI-00030	FAST	Allows a future invention of an optimization
AI-00031	NEUT	(I think it should raise a constraint error)
AI-00032	FAST	Appendix F requirements should be updated.
AI-00034	NEUT	This is not related to speed, but to why tasks don't know their own names, which will be fixed by 9X.
AI-00035	NEUT	syntax only
AI-00037	FAST	encourages generics with the consequences that generic optimizations will eventually be invented
AI-00038	NEUT	made generics bind like a renames clause
AI-00039	NEUT	eliminated a contradiction
AI-00040	NEUT	eliminated a contradiction
AI-00045	NEUT	allows non-static priorities
AI-00046	SLOW	Part 2 is wrong because it will have to be programmed around. There is no reason to make the name function work differently on temporary files.
AI-00047	NEUT	standardization
AI-00048	NEUT	standardization COULD BE FASTER: if it emphasized that opening files should be as quick an operation as possible. For example, when you put a non-null string as the fourth argument of OPEN on one Vax/VMS Ada compiler, it takes several seconds to open the file, because of the RMS interpreter Other Vax/VMS languages are allowed to open files with the string pre-interpreted, with virtually instant file opens. And that string is the only way to control the size and shareability of the file under VMS.
AI-00050	NEUT	standardization of skip line
AI-00051	NEUT	standardization of integer input
AI-00099	NEUT	changed nothing
AI-00103	FAST	without this, fewer computations could be done at compile time
AI-00113	NEUT	changed nothing
AI-00120	NEUT	does not affect run time
AI-00128	SLOW	REVERSE THIS RULING so more expressions are considered static.
AI-00132	NEUT	since nobody implemented it, forbidding it does not change anything
AI-00137	SLOW	This is slower, but possibly not needed.
AI-00138	NEUT	since nobody implemented it, this is not changing anything
AI-00139	NEUT	the comments in the commentary should be in the ACVC test
AI-00143	SLOW	Not making the bounds model numbers is a problem in accessing all the bits in a word. This results in programming around the problem to take advantage of every binary number. The Standard should be changed to allow access to the model numbers, at compile time and at run time.
AI-00144	SLOW	(see AI-00143)
AI-00145	SLOW	the ramification should have said: compute the value at compile time whenever possible
AI-00146	NEUT	nothing changed
AI-00147	SLOW	-1.0 MUST BE A MODEL NUMBER on a two's complement or a non-binary machine; also see (AI-00143). If this takes a pragma to tell the compiler whether the arithmetic is one's complement, two's complement or non-binary, then so be it, but let the programmer have access to every number that fits in that address.
AI-00148	SLOW	First of all, -1 is a static constant, not an expression, and second of all, unary negation should be static. The arguments given in this commentary apply to the range 1..10, just as well as they apply to the range -1..10. These interpretations need to be changed. The same problem happens with

fixed points because -x means -1 * x instead of 0.0 - X or additive inverse. This is a real turn-off for any programmer, designer, or mathematician who attempts to use Ada. Please fix it.

AI-00149 SLOW Elaboration order is a hard topic. It would be nice if it could be resolved at compile time. The language should require warnings at compile time for programs whose elaboration order "could" result in contradictions. This is a conceivable warning, as opposed to warning which programs "will" result in contradictions. This is a conceivable warning, as opposed to warning which programs "will" result in contradictions. The warning would identify programs who "with" each other and call each other during elaboration of global data or package initialization.

AI-00150 SLOW The commentary forbids

```
type a is access string (1..10):
x: a;
x := new string (2..11);
```

This is very slow. There is no difference between 1..10 and 2..11 from a realtime point of view. This view of constraints if a key reason that people give why Ada is "hard" to program in. Giving up the distinction between 2..11 and 1..10 would make Ada programs do less constraint checking, give programmers fewer unreasonable restrictions on simple constructs and run faster. No optimization nor software engineering benefits have arisen from substrings beginning other than at 1.

If all arrays and slices begin at 1, there are a lot of uncomfortable rules and expensive checks that don't have to be done. These checks don't find bugs in the design or code (other than bugs because the program does not follow those rules). These ruies do not make more readable or more reliable software. As a matter of fact, they bring in to Ada a level of unreliability because more testing is required of Ada programs which do slices than most other languages, because Ada programs must be further tested, after they work perfectly, to verify they still work when the slice does not begin at 1.

AI-00151 SLOW The instantiation is also compiled, the case statement is fully known at compile time. Each instantiation MUST be optimized separately, in order to run fast. Conditional compilation must remove code and other optimizations must come into play, which require each instantiation to be optimized separately. Since execution speed is the main objective, this ALRM restriction should be removed and the ALRM should be modified to say that each instantiation is optimizable separately.

AI-00153 NEUT no change

AI-00154 NEUT no change

AI-00155 NEUT syntax only

AI-00157 NEUT syntax only

AI-00158 SLOW the requirement for main programs is nonsense; the ALRM should allow "execution" of packages (by elaborating them); the concept of main programs is completely unnecessary

AI-00163 SLOW to allow many more expressions to be static, type conversion should be allowed by the ALRM as rule (g) in 4.9.; this is essential for programs to run fast, which use type attributes and predefined constants, or which use fixed point constants at all.

AI-00167 SLOW This inconsistency arose when the task activation verb was deleted from the language, without making the other adjustments that followed from it. This

		is part of the reason why tasks don't know their names. Now for the sake of orthogonalization, we are stuck with such requirements as this, to have a task existing which does not fully exist.
AI-00169	SLOW	Ada should make no distinctions between null arrays and missing arrays. This slows execution by requiring additional run time "dope" vector entries to track where the constraint starts, whether it is null, etc.
AI-00170	SLOW	A renames clause should not be forbidden in this case, it should just generate a constraint check. Think about what the programmer has to do to program around this.
AI-00172	SLOW	Page terminators should be redefined to appear anywhere, not necessarily only after a carriage return. Text_io is not a "good" or "fast" interface for text. Besides, it does not handle escape sequences, the most common way of controlling text devices such as printers, keyboards and monitors.
AI-00173	FAST	quicker completion is better
AI-00177	FAST	any allowance of "others" speeds up potential optimizations which can do the initialization at compile time instead of run time; or better yet, they can avoid the initialization by intelligently using the known initialization value to prevent generating code at all, based on its actual usage
AI-00179	FAST	"fore" is machine dependent
AI-00180	FAST	no change
AI-00181	SLOW	Now a programmer must go through many contortions to get certain expressions evaluated, which should be static after 9X.
AI-00186	SLOW	This would not have been an issue, if type conversions were considered to be static operators, which they should be in almost all cases. Type conversions, including unchecked conversions, should almost never generate code.
AI-00187	FAST	This encourages "renames" and their comcomitant readability and optimizability.
AI-00190	SLOW	change this interpretation immediately. Static expressions must be allowed to have generic formal types. Generic Instantiations must compile and optimize at instantiation time, in order for generic instantiations to run as fast as non-generic instantiations.
AI-00192	NEUT	The Ada manual s/b updated to include these restrictions
AI-00193	NEUT	syntax only
AI-00195	NEUT	nothing changed
AI-00196	NEUT	nothing changed
AI-00197	SLOW	type priorities should be defined in package system. This makes machine dependencies easier to find mechanically which makes software reuse less costly.
AI-00198	NEUT	nothing changed
AI-00199	SLOW	Most of this interpretation is correct and fast. However, the one sentence that slows down execution is "A subsequent recompilation of the body will replace the library unit in the program library, but NOT THE PRAGMA, which can still be obeyed". This trivializes the "pragma inline", a potent source of optimizations. The ALRM should be modified (10.4) to say that pragma inline is part of the compilation AND PART OF THE BODY TO WHICH IT APPLIES.
AI-00200	SLOW	the part that is "slow" is the unanswered restriction that "implementations .. are not allowed .. to implicitly recompile " a unit. Change the ALRM to not forbid this.
AI-00201	SLOW	The lack of answer to the question of how accurate is a "delay" statement

		executed, is causing problems. The answer given in the Commentary was that it is "significantly worse" than system.tick, then "justification would be required in terms of AI-00325". If it does not have to be updated with the accuracy of system.tick, then we need a variable in system to say how accurate it is, so that hard realtime systems can be programmed. There needs to be a way of objectively measuring how we are doing.
AI-00205	NEUT	nothing changed
AI-00209	FAST	Again, this encourages compile time rather than run time expression evaluation, which is a powerful optimization.
AI-00214	NEUT	Non-simple names here are violations of structured programming. However, this does not affect runtime.
AI-00215	NEUT	nothing changed
AI-00217	NEUT	nothing changed
AI-00219	SLOW	The ALRM must be changed to indicate that concatenation image, all attributes whose values can be computed at runtime, trigonometric or min/max functions or any functions or attributes that are declared to have no sid effects can all be used in static expressions.
AI-00225	NEUT	nothing was changed
AI-00226	NEUT	nothing was changed
AI-00231	NEUT	nothing was changed
AI-00232	NEUT	nothing was changed
AI-00234	FAST	all strings should start at 1
AI-00235	NEUT	no runtime effect]
AI-00236	SLOW	The ALRM should be modified to require no pragma elaborate for missing bodies. The "with P" in the example already required that the task be started before Q's elaboration.
AI-00237	NEUT	nothing changed, but see AI-00236 above
AI-00239	SLOW	non-graphic characters should be equally readable to graphics
AI-00240	SLOW	A range attribute should be allowed to define a type, if the range is known at compile time
AI-00242	NEUT	syntax only
AI-00243	NEUT	no change
AI-00244	SLOW	the constraint was probably static, after the 9x revision to the definition of static, and therefore the record should be considered static
AI-00245	SLOW	this example should be allowed and all type conversions should be treated as static and generate no code, where it is possible to do so, which is probably everywhere.
AI-00247	SLOW	Universal defaults should exist on each implementation and it should be legal to leave out the FORM. In addition, these defaults should be accessible for the user to set.
AI-00251	SLOW	Types derived from generic formal types MUST BE STATIC, if the original was static. This should continue to be true after 9x expands the list of expressions to be considered static.
AI-00257	FAST	allows more inlining
AI-00258	FAST	implementation does not have to store the extra value, thus saving execution time at the expense of flexibility
AI-00260	NEUT	syntax only
AI-00261	NEUT	syntax only
AI-00263	NEUT	COULD BE FASTER; the ALRM should mention that they should be optimized away
AI-00265	SLOW	none of these should raise constraint errors, since they are implicitly

		convertible. Type conversions should also be allowed among these subtypes. The place where a subtype starts should have no bearing on its compatibility with other subtypes. The string example should be treated no differently from the integer array examples. Implicitly subtype conversions should be performed on aggregate evaluations, exactly as they are for assignment statements. By making this universal, all slices can start at 1 internally, and no run time code need check for this kind of constraint.
AI-00266	NEUT	syntax only
AI-00267	SLOW	The expression (X+1) should be considered static here and be computed and detected out of range at compile time.
AI-00268	FAST	The less run time checking, the better.
AI-00276	SLOW	Rendezvous does far too much checking already. Tasks should do most of their own checking,there must be fewer states (terminated vs aborted vs could not be started, etc), and the task must determine its own status, not the rendezvous code. A rendezvous must be implementable with virtually no time elapsed for checking the many rules we now have. (Therefore we need fewer rules).
AI-00279	NEUT	standardizes exception names
AI-00282	NEUT	nothing changed
AI-00286	NEUT	syntax only Ada 9X should adapt the resolution that generics do less checking at definition time and more at instantiation time.
AI-00287	NEUT	syntax only
Ai-00288	FAST	needed to prevent priority inversion
AI-00289	NEUT	syntax only. No need to forbid it, though
AI-00292	NEUT	syntax only
AI-00293	NEUT	syntax only This confirmation is wrong, by the way. This makes the syntax "soft" and allows two ways of expressing each aggregate.
AI-00294	NEUT	syntax only This ramification is wrong, by the way. There needs to be a standard pragma (say, ALLOCATE) of allocating memory to different memory pools, the location, size and speed of which must be expressible by another pragma (say POOL). In addition to the ALLOCATE pragma, there must be a pragma which requests the compiler to optimize the allocation of the remaining variables to the best usage memory pool, after all ALLOCATE pragmas have been executed.
AI-00295	NEUT	syntax only
AI-00298	NEUT	syntax only
AI-00300	NEUT	syntax only
AI-00305	FAST	Because now programmer can branch there
AI-00306	NEUT	syntax only
AI-00207	NEUT	standardizing which exception is raised
AI-00308	SLOW	In this case, as in most cases, the discriminant should be considered static, and the message should be given at compile time and the program should not execute.
AI-00310	FAST	It is good to remove most of the remaining restrictions on the use of OTHERS, so that more use of it can be made, because it can usually be optimized in a way that no code need be generated.
AI-00211	FAST	There should be few restrictions, if any, on null strings.
AI-00312	FAST	Sometimes slow. But since all extended static expressions will be evaluated

		at compile time, usually fast.
AI-00313	SLOW	Since the purpose of the type conversion of an aggregate is to adjust the bounds, forbidding these type conversions not only forces the implementor to generated run time checking, but also restricts the programmers freedom with no corresponding gain in functionality or safety. No code at all need be generated for aggregate subtype conversions, if all arrays are stored as if they started at 1.
AI-00314	NEUT	no change
AI-00316	NEUT	no change
AI-00319	NEUT	no change -- the check is required for consistency
AI-00320	SLOW	Since the line and column cannot be changed between shared files, it is impossible and inconsistent for reading one file to cause end of file on the other one. (Recommendation, third paragraph). End of file is a function of the file object, not the external file. The same with page, column, line, column, etc.
AI-00321	NEUT	syntax only
AI-00322	NEUT	syntax only
AI-00324	FAST	but it introduces run time errors into programs
AI-00325	FAST	need the # of machine dependencies to get much smaller
AI-00328	SLOW	Almost all checking should be done at instantiation time
AI-00330	NEUT	syntax only
AI-00331	FAST	but like AI-00324, it introduces run time errors into programs
AI-00332	NEUT	syntax only
AI-00336	NEUT	syntax only
AI-00339	FAST	will make it easier to implement the 8-bit codes
AI-00343	SLOW	does not allow filling every value in a word
AI-00350	NEUT	syntax only
AI-00354	SLOW	the language needs pre-elaboration; (also incomprehensible)
AI-00355	SLOW	redefine language to allow compile time elaboration
AI-00356	FAST	
AI-00357	FAST	could be faster: add append capability without copying
AI-00358	FAST	(but incomprehensible)
AI-00362	SLOW	reduces ability to work with bit strings
AI-00365	NEUT	no change
AI-00366	FAST	Ada SHOULD start encouraging more source portability
AI-00367	NEUT	syntax only
AI-00370	NEUT	syntax only
AI-00371	NEUT	syntax only
AI-00374	FAST	(but contradicts "Dear Ada")
AI-00375	NEUT	no change
AI-00376	SLOW	(and stupid)
AI-00379	NEUT	syntax only
AI-00384	SLOW	must expand at instantiation time
AI-00387	FAST	standardizes exceptions
AI-00388	NEUT	syntax only
AI-00396	NEUT	documentation only
AI-00397	SLOW	but necessary
AI-00398	NEUT	syntax only
AI-00405	NEUT	allows better compile time expression evaluation
AI-00406	NEUT	could be faster: if elaboration could be checkable at compile time
AI-00407	SLOW	subtypes s/be extremely simple; and types s/fit in fewer bits

AI-00408	FAST	the ONLY concern in generics is equal speed to non-generics
AI-00409	FAST	makes more expressions static
AI-00412	NEUT	syntax only
AI-00418	NEUT	syntax only (note: not FAST, since elabs s/be optimizable away)
AI-00422	NEUT	does not restrict application from any bit allocation desired
AI-00425	FAST	makes more expressions static
AI-00426	NEUT	(should be a consequence, not a binding interpretation)
AI-00430	NEUT	no change
AI-00431	FAST	packed boolean operators must be EXTREMELY fast and simple
AI-00441	NEUT	no change
AI-00444	NEUT	no change
AI-00446	SLOW	change language to allow opt. away of rendezvous exceptions
AI-00449	NEUT	syntax only
AI-00455	NEUT	no change
AI-00464	NEUT	no change
AI-00466	NEUT	I/O only
AI-00467	SLOW	Deltas should follow the rules we learned in grade school for precision of numbers
AI-00468	NEUT	syntax only
AI-00471	NEUT	syntax only
AI-00475	NEUT	no change
AI-00483	NEUT	syntax only
AI-00486	NEUT	syntax only
AI-00493	NEUT	syntax only
AI-00502	NEUT	no change (spelling error: not corrected as it says)
AI-00503	NEUT	no change
AI-00508	NEUT	no change
AI-00511	NEUT	syntax only
AI-00516	FAST	syntax only

ALLOW MORE EXPRESSIONS TO BE CLASSIFIED AS STATIC

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 4.9

PROBLEM:

The precise definition of "static" is used in Ada rather than the loose concept of known-at-compile-time. There are specific semantics that require something static. The problem is that the 1983 specification is so conservative that it becomes very difficult to write some numerical applications.

The most severe problems occur in developing generic numerical library packages. The desire is to have efficient code generated for each precision. There is a possibility of code sharing within a floating point hardware precision. There is an explicit desire to not have code sharing between different floating point precisions. For example :

```
generic
 type P is digits <>;
 type V is array(INTEGER range <>) of P;
package NUM is
 function DOT(V_1, V_2 : V) return P;
end NUM;

with SYSTEM;
package body NUM is
 function DOT(V_1, V_2 : V) return P is

  -- for V_2'FIRST use V_1'FIRST;
  -- would save on offset computations
  -- additional length clause
  -- changes local array bounds

  type DOUBLE_P is digits -- minimum of 2*P'DIGITS and SYSTEM.MAX_DIGITS
        (12+SYSTEM.MAX_DIGITS) - abs(12-SYSTEM.MAX_DIGITS); -- OK
        -- (2*P'DIGITS+SYSTEM.MAX_DIGITS) - abs(2*P'DIGITS-SYSTEM.MAX_DIGITS)
        -- will not work unless  pragma ONE_PRECISION  could make P'DIGITS
        -- static
        -- INTEGER'MIN( 2*P'DIGITS, SYSTEM.MAX_DIGITS) would be better

   SUM:       DOUBLE_P := 0.0;
 begin
```

```
    for I in V_1'RANGE loop
        SUM:=SUMDOUBLE_P(V_1(I))DOUBLE_P(V_2(I-V_1'FIRST+V_2'FIRST));
    end loop;
    return P(SUM);
  end DOT;
end NUM;
```

IMPORTANCE: IMPORTANT

This is an inhibitor to the development of numerical library packages. With more than 27 Ada commentaries in the rather small section 4.9, it is important to rework this section. Upward compatibility must be maintained, thus more needs to be defined as static, certainly not less.

CURRENT WORKAROUNDS:

Struggle to find workarounds. It is desired to do the inner loop in higher machine precision:

POSSIBLE SOLUTIONS:

Broaden the definition of static to the practical limit. The goal is to allow more expressions, attributes and array bounds to be static.

BASE ATTRIBUTE OF A GENERIC FORMAL TYPE

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 4.9(11)

PROBLEM:

Most Numerically Intensive Computing (NIC) software is and will be designed to be generic with respect to one or more floating-point types (among other generic parameters). These types are typically used within generic packages to parameterize the parameter and result types of their exported subprograms, thereby gaining reusability. Usually it is appropriate to use, within the implementation of those subprograms, local (working) variables having the same precision. The most straightforward and obvious way of accomplishing that is to use a generic formal type directly to declare the working variables, as is done for W in

```
generic
        type FLOAT_TYPE is digits <>;
package GENERIC_MATH_FUNCTIONS is
        ...
        function F (X : FLOAT_TYPE) return FLOAT_TYPE;
        ...
end;

package body GENERIC_MATH_FUNCTIONS is
        ...
        function F (X : FLOAT_TYPE) return FLOAT_TYPE is
                W : FLOAT_TYPE;
        begin
                ...
                W := ...
                ...
                return ...
                ...
        end;
        ...
end;
```

The problem is that a user might instantiate this generic unit with a generic actual (sub)type having a range constraint, as in

```
        subtype SPEED is FLOAT range 0.0 .. 100.0;
        package MY_MATH_FUNCTIONS is
```

```
        new GENERIC_MATH_FUNCTIONS (SPEED);
    use MY_MATH_FUNCTIONS;
```

Whereas it is usually appropriate for the parameter and result types of the subprograms in the generic unit, like those of F, to inherit the range constraints of the generic actual subtype, it is often disastrous for the working variables, like W in the body of F, to inherit them. Even when the parameters and results of F always satisfy the range constraints, it is unreasonable to assume that the values of the internal, working variables of F do, especially since the range constraints could be arbitrarily narrow. Thus, subprograms like F can too easily be rendered useless by their users if this implementation technique is employed by their implementors, destroying their reusability.

A method is required for a generic unit to retrieve, and use directly, the base type of the generic actual (sub)type associated with one of its generic formal parameters that is a generic formal type. The present language prevents this in two ways: by not allowing the use of FLOAT_TYPE'BASE as a type mark [RM 3.3.3(9)] and by declaring attributes of gen-eric formal types not to be static [RM 4.9(11), 12.1(12)].

A closely related problem is the inability to express precision relationships in generic units--for example, that while one variable only needs the precision of the generic actual type, another needs twice that precision. Certain critical steps of numerical algorithms, such as an accumulation of many items, or an argument reduction, require extra precision in some well-defined relative sense. The required control is lacking once again because generic for-mal types are not static and hence neither are their attributes.

IMPORTANCE: ESSENTIAL

A solution to the problem described above is ESSENTIAL for NIC software. Without a viable solution that is much better than the workarounds discussed below, such software will pay an unacceptable price in terms of lack of robustness, inefficiency, and risk of error, or alternatively Ada will not be used for such software.

CURRENT WORKAROUNDS:

One solution to requesting adequate precision for working variables is to ignore the generic formal parameters and always request maximum precision for them. This is unattractive because the extra precision (beyond what is logically needed, and what could be obtained if we but had a way) might carry an unacceptable cost. For example, double-precision instructions might be slower than single-precision instructions, or the occupation of twice as much storage as is required may be unaffordable (e.g., when large temporary arrays are required).

The use of maximum precision for the higher of two related precisions suffers the same problems.

Another potential workaround is captured by the following technique:

```
package body GENERIC_MATH_FUNCTIONS is
    ...
    function F (X : FLOAT_TYPE) return FLOAT_TYPE is
    begin
        case FLOAT_TYPE'BASE'DIGITS is
        when 1 =>
            declare
                type WT is digits 1;
                W : WT;
```

```
                    begin
                        ...
                    end;
                when 2 =>
                    declare
                        type WT is digits 2;
                        T : WT;
                    begin
                    ...
                    end;
                ... -- etc., for each value up to some
                    -- reasonable maximum
                end case;
            end;
            ...
        end;
```

This technique is ugly, wasteful, and prone to error. Nevertheless it, or a close variant in which the number of cases is reduced by lumping some of them together, is what is most commonly followed today.

POSSIBLE SOLUTIONS:

One should be able to write:

```
package body GENERIC_MATH_FUNCTIONS is
    ...
    function F (X : FLOAT_TYPE) return FLOAT_TYPE is
        subtype WT is FLOAT_TYPE'BASE;  -- (*)
        W : WT;                         -- (*)
    begin
        ...
    end;
    ...
end;
```

Or, in place of the two lines marked "-- (*)", it would be desirable to write

```
AT_TYPE'BASE,
```

Thus W will generally have the same precision as the base type of the generic actual (sub)type without its restrictive range constraints. It is highly desirable that W be declarable as a subtype so that, for example,

```
W := X;
```

can be written (i.e., so that W and X have the same base type and can be mixed in expressions without explicit conversions).

The problem of two types related through their precisions could be addressed by allowed language such as

```
type WT1 is digits FLOAT_TYPE'BASE'DIGITS;
type WT2 is digits 2*FLOAT_TYPE'BASE'DIGITS;
```

(In general, of course, since 2*FLOAT_TYPE'BASE'DIGITS might exceed SYSTEM.MAX_DIGITS, other techniques would have to be brought into play to avoid a possible error. Such techniques are known and are not germane to this revision request.)

One language change that is required is that attributes (or at least some attributes) of generic formal types should be considered static, or at least that they should be usable as operands in static expressions. Another, independent, change is that the 'BASE attribute should be able to stand by itself as a substitute for a type mark. Another context where this is desirable, besides what was shown above, is this: it might be appropriate for the designer of a generic package like GENERIC_MATH_FUNCTIONS to use FLOAT_TYPE for the parameter types of the included functions but FLOAT_TYPE'BASE for their result types. Who is to say, after all, that the results of a function satisfy the same (arbitrarily narrow) range constraints as its arguments? In general, they don't. The ability to shed range constraints from the results of a function, but otherwise keep the same floating-point precision as the arguments, mimics what happens with the predefined arithmetic operators (actually, the analogy would be even better if one shed the range constraints during parameter association as well).

NOTE: We note that it might be undesirable that the ability to use 'BASE for type declarations is allowed for arbitrary types including, for example, array types. But, apart from floating-point types, the ability would be very useful for fixed-point types and integer types as well.

EXPRESSIONS LIKE a<b<c ARE NOT ALLOWED

DATE: September 3, 1989

NAME: Erland Sommarskog

ADDRESS: ENEA Data AB
Box 232
S-183 23 TABY
SWEDEN

TELEPHONE: +46-8-7922500
E-mail: sommar@enea.se

ANSI/MIL-STD-1815A REFERENCE: 4.4, 4.5.2

PROBLEM:

Expressions like a < b < c are not allowed, but has to be written a < b AND b < c. This is less clear and not always immediately understood. Particularly this is true when b is a long expression, readability suffers.

IMPORTANCE: IMPORTANT

Undoubtedly a nice-to-have feature, but one you would expect to appear in any programming language, albeit it does not. It is my firm belief that the simple reason is that Algol-60 didn't have it, and language designers since then have had higher aims and over-looked this nice little detail.

CURRENT WORKAROUNDS: See above.

POSSIBLE SOLUTIONS:

Change the production in section 4.4(2) from

```
relation ::=
        simple_expression [relational_operator simple_expression[...
to
relation ::=
        simple_expression {relational_operator simple_expression{...
```

The semantic interpretation of an expression like

```
        A rel_op B rel_op C rel_op D ...
```

would be a conjunction of each pair, thus the above would be the same as:
A rel_op B AND rel_op C AND C rel_op D ....
The order in which the expression is evaluated not be defined by the language.

However, the language would require that no simple_expression is computed more that once. Thus in the case of:

a < f(p) < c
f should be called (at most) once.

In the case of overloading, the requirement must be that all involved relational operators returns the predefined type boolean. (One could imagine that a < b returned a value of some type and b < c some other type, and there was an AND operation defined for these types. This is a pathological case I am reluctant to say that the language should permit it.)

The application of parenthesis to enforce that some part of the relation be computed first would not be possible. The interpretation of:

a < (b < c)

would be the same as today, and thus only be legal if a is of type boolean. (More generally: There is a "<" for a:s type on the left and the type of b < c on the right.)

For additional references to Section 4. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

NUMBER	TITLE	PAGE
0045	OVERFLOW AND TYPE CONVERSION	3-28
0070	USER DEFINED ASSIGNMENT	7-10
0250	NULL SPECIFICATION FOR NULL RANGES AND RAISING EXCEPTIONS ON NULL RANGE ASSIGNMENT ERRORS	3-116
0251	APPEAR TO BE FUNCTIONS IN THE SOURCE	3-151
0354	PHYSICAL DATA TYPES	3-101
0365	IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (I)	3-121
0367	NATIONAL LANGUAGE CHARACTER SETS	2-11
0369	ADA SUPPORT FOR ANSI/IEEE STD 754	3-103
0391	CLUMSY SYNTAX FOR REPRESENTING BASED NUMBERS	2-23
0412	OVERLOADING OF EXPLICIT EQUALITY "="	6-102
0438	HANDLING OF LARGE CHARACTER SET IN ADA	2-12
0556	USE OF PARENTHESES FOR MULTIPLE PURPOSES	2-18
0636	FLOATING POINT NON-NUMERIC VALUES ("NAN'S")	3-167
0637	THE STATUS OF FLOATING-POINT "MINUS ZERO"	3-169
0643	GARBAGE COLLECTION IN ADA	3-237
0702	HEAP MANAGEMENT IMPROVEMENTS	3-241
0704	MAKE EVERY BIT AVAILABLE TO THE APPLICATION PROGRAMMER	2-19
0704	GENERIC FORMAL EXCEPTIONS	12-19
0724	IMPLICIT CONVERSIONS AND OVERLOADING	8-58

0745 INTELLIGENT STRONG TYPING 3-67

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 5. STATEMENTS

LABEL AND PROCEDURE VARIABLES

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 5.1, 5.9, 6

SUMMARY:

The requirement is for Ada to support some form of restricted label variables. The requirement comes from programs simulating state machines, machine language emulators, etc. While a label variable can be emulated reasonably efficiently using an integer variable and a huge case statement, the program becomes quite unreadable, and unmaintainable. We propose a very restricted form of label variable which should be easy to implement as well as efficient. It also solves another technical problem in the emulation of some other high-level languages within Ada.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

In the building of an emulator for a machine language, a need arises for label variable. For example, to simulate a procedure call in a machine language, on wishes to push a "return address" onto a stack, followed by a jump to the body of the procedure. When the procedure wishes to return, it pops the "return address" from the stack, and jumps to that location.

In a high-level language such as Ada, a program point can be represented by a (constant) Ada label, but there is no way to push such an object onto a stack, nor jump to an object pooped off of such a stack. The usual solution is to "encode" the program point into an enumeration variable, in which every item is in a one-to-one correspondence with an Ada label. During the procedure call, the "code" for a return label is pushed onto the stack, while during the return, the code is pooped from the stack and a case statement is used to "decode" the code into an actual jump to a label.

While this solution works, is "reasonably" efficient, it is not satisfying. The program point labels are only visible within the inner context, and therefore it is difficult to package up the abstraction.

The workaround solution requires a lot of essentially redundant code, and this code can become extremely unwieldy--e.g., if there are hundreds or thousands of return points.

We propose that Ada be extended with labels that .re not complete program points, but are only machine code addresses. We recommend that the syntax and semantics be so restrictive that these labels can only be sued in the same environment that they could be used had they been label constants. In this sense, we are adding no new semantic power to Ada, because these label variables are no more powerful than the corresponding enumeration-coded variables, but they are a lot easier to deal with as a programmer, and perhaps 2-3 times more efficient at run-time.

In particular, we are not recommending the very expensive label variable mechanism of PL/I for Ada.

Label variables of this sort can also be valuable in solving a problem of procedure variables. While the inclusion of general procedure variables into Ada is beyond the scope of this request, a label variable can be used as a cheap form of procedure variable for procedures without any arguments which only call other procedures. Procedures which have no arguments and which call other procedures as the last thing they do before they themselves return can simply jump to this last procedure instead of calling it. This saves both space and time. Space is saved on the stack, because a return point need not be pushed. Time is saved, because when the procedure returns which was jumped to, it will return not to the procedure which called it, but directly to that procedures which must be called in the "tail" are better jumped to than called [Steele].

It is difficult, if not impossible, to generally simulate tail-calling behavior in a language like Ada that doesn't already have it [Bartlett]. This is because argument/result handling for tail calling requires quite a bit of subtlety to handle correctly. However, if a procedure has no arguments and no result (parameterless procedure), then there are no problems, and the implementation is extremely easy.

We thus request that Ada require parameterless procedures which are called from a "tail position" to be "tail-called". This would allow Ada to simulate certain control structures that would otherwise require an enormous amount of work.

The net result of this request is that labels and parameterless procedures become almost the same.

CURRENT WORKAROUNDS:

The Workaround is given above.

NON-SUPPORT IMPACT:

Certain emulations and control structures are currently impossible to simulate within Ada without simply "flattening" the program and treating Ada as an assembly language. Putting label and parameterless procedure variables into Ada would enable these emulations and control structures to be handled using the more normal Ada abstraction mechanisms.

POSSIBLE SOLUTIONS:

Discussed above.

DIFFICULTIES TO BE CONSIDERED:

The major difficulty is in keeping the proposal simple so that there are very minimal implementation problems. Trying to handle full program-point labels or full procedure labels would create an enormous amount of baggage, which would mean that the simplest cases would be very slow. We are after a minimal capability that can be efficiently implemented.

This proposal is a direct violation of Steelman requirement 6G.

REFERENCES:

Bartlett, Joel F. Scheme->C: a Portable Scheme-to-C Compiler. Digital Equipment Corporation Western

Research Laboratory Research Report 89/1, Jan. 1989.

Steele, Guy L. RABBIT: a Compiler for Scheme. AI Memo 452, MIT AI Lab., May 1978.

ELIMINATION OF GOTO CONSTRUCT

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 5.1(7), 5.9

PROBLEM:

Goto's encourage bad programming habits. If this is a desirable goal then we should be encouraging everyone to write in FORTRAN, COBOL, or assembly language instead of Ada.

IMPORTANCE: IMPORTANT

Goto's should never have been allowed in the standard in the first place. Ada has more than enough looping, branching, exception handling, and other control structures as it is without allowing Goto's. There is no justification for the Goto other than nostalgia.

CURRENT WORKAROUNDS:

Good programming discipline, coding standards, code audits, code checking utilities.

POSSIBLE SOLUTIONS:

Eliminate Goto's from the language.

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed. On the other hand, only badly-written code will fail to compile, so this is not great loss.

ASSIGNMENT FOR LIMITED PRIVATE TYPES

DATE: June 29, 1989

NAME: Larry Langdon

ADDRESS: Census Bureau
Room 1377-3
Federal Office Bldg 3
Washington, DC 20233

TELEPHONE: 301-763-4650
E-mail(temporary): langdonl@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 5.2, 7.4.4

PROBLEM:

A type, T, which is not inherently limited (i.e., its full type definition does not contain limited sub-components) is generally declared limited private when either or both of the standard equality test or assignment of objects of such types is inappropriate. For such types it is possible to redefine the equality test as necessary. No such definition is possible for the assignment "operator" (:=). Such definition is frequently desired (see "Motivating Examples" following).

IMPORTANCE: IMPORTANT

If this feature is not included, programs will continue to be less readable than they might be, using procedure calls as workarounds whenever an assignment-like operation is used on limited-type objects. We will continue to be unable to write aggregates that contain limited- type objects, or to write generic formal objects with default values for this class of types.

It also forces us to distinguish between generic data structures whose components are limited and those whose components are not, meaning in many cases a doubling of effort (one generic package/subprogram with a limited-type parameter and one with non-limited) of building generic libraries.

The tendency of the current state of the language is to discourage the use of limited types. Instead we use non-limited types and rely on the consistent self-discipline of programmers to use these types correctly. Of course, the reason we find so much emphasis on the use of high-level languages today is that this kind of reliance tends to be a risky and costly practice.

CURRENT WORKAROUNDS:

Use of procedure calls (SET, ASSIGN, etc.). This is ugly, decreases the readability of the code, and has been found to be error-prone (What was the name again? Which argument goes first? etc.). It also requires the introduction of extra statements when the desired effect could have been achieved by object initialization (":=") or aggregate assignment.

POSSIBLE SOLUTIONS:

Allow user-defined assignment (:=) for limited types (not task types or types containing them). This re-definition of assignment is intended to apply only to the use of ":=" where it is not now allowed. In particular, it should not apply to the copy-in/copy-back of subprogram parameters. It should, however, apply to aggregates, to initial values for record type and object declarations, and to allocators. Permissible user-defined assignment contexts would include, for example, an assignment procedure for a limited private type T, declared within the package defining T. Another would be an assignment procedure for an array of T, declared outside this package (and using the assignment for T defined in the package) in the obvious way.

Any procedure re-defining ":=" for a type T should be required to have two parameters, both of type T. The first should represent the target object and be of mode "out" or "in out" (the latter possibility being desirable, for example, to allow garbage collection). The second should represent the source and be of mode "in" or "in out".

POSSIBLE SYNTAX

One possible syntax for such a definition might be as follows, using T to represent the limited private type in question:

```
        procedure ":="(target : out T; source : T); -- in the specification
and
        procedure ":="(target : out T; source : T) is -- in the package body
                ...
        end ":=" ;
```

Another possible syntax might use a pragma to associate an ordinary procedure with the above argument structure with assignment:

```
                procedure xyz(target : out T; source : T);
                pragma assignment(asg_proc=>xyz,asg_type=>T);
```

(where the "asg_type" argument might be thought of as benign redundance).

ADDITIONAL ISSUES

* It might be deemed appropriate to allow the "source" (parameter 2) to be of a different type. This would allow the use of the assignment statement for type conversions, for example string to VSTRING (see example 1, below).

* Note that since T is limited, any type containing T will be also, so that the re-defined ":=" will not require re-consideration of aggregates containing values of type T. (Nevertheless, if it were deemed possible to extend re-definition of ":=" (and "=") beyond limited private types, it would be useful to have ":=" inherited by containing types)

* The restriction of assignment re-definition to limited types may not be absolutely necessary. It is there that the need is greatest, since there is no assignment at all. Also, it removes certain concerns (see the preceding discussion of aggregates). We know of no reason why it could not be done for all types.

MOTIVATING EXAMPLES

1. Variable-length strings

Suppose you define a variable-length string as:

```
subtype INDEX is integer range 0..max_allowable_string_length;
type VSTRING(maxlen : index) is
        record
        len : index := 0;
        val : string(1..maxlen);
        end record;
```

with the semantics that V.LEN is the current length of the VSTRING, V, and V.VAL(1..V.LEN) is its current contents (with the rest of V.VAL being irrelevant). Assignment for VSTRINGs should depend on current values: Two VSTRINGs having different maximum lengths should not prevent the assignment of one to the other. A user-defined assignment for VSTRINGs might look like:

```
procedure ":="(s : out vstring;  t : vstring) is
begin
        if  t.len > s.maxlen  then  raise vstring_overflow;  end if;
        s.len := t.len;
        s.val(1..t.len) := t.val(1..t.len);
end;
```

2. Other variable-length objects

The reasoning of example 1 applies also to other abstract data types implementing variable-sized objects. As one such example, you might choose to implement arbitrary-precision integer arithmetic using the type:

```
subtype index ... (as above)
type int_array is array(positive range <>) of integer;
type big_integer(maxlen : index) is
        record
                len : index;
                negative : boolean;  -- arithmetic sign of the big_integer
                val : int_array(1..maxlen);
        end record;
```

For reasons analogous to those in example 1 above, the assignment should have semantics like:

```
procedure ":="(a : out big_integer;  b : big_integer)  is
begin
        if  b.len > a.maxlen  then  raise  big_int_overflow;  end if;
        a.len := b.len;
        a.negative := b.negative;
        a.val(1..b.len) := b.val(1..b.len);
end;
```

3. Types implemented as access variables

An abstract data type implemented as an access type will generally find the standard assignment semantics inappropriate. For example, consider:

```
type DSTRING is access string;
     s, t : dstring;
```

If a DSTRING is considered as an abstract "dynamic" string, defined in a package, then s and t are supposed to represent dynamic strings to the user of the package. The fact that they are implemented as access types should not be relevant (or possibly even known) to the user. Therefore,

```
s := t;
```

should mean that the dynamic string s is given a value equal to that of t, not that s now points to t. In other words, the code sequence (which assumes the existence of a string to DSTRING conversion function, ds):

```
t := ds("  ");
s := t;
t := ds("hi there");
if  s /= ds("  ")  then  raise oops;  end if;
```

should not cause oops to be raised. You should not have to (or be able to) refer to s.all. A plausible semantics for DSTRING assignment might be:

```
procedure ":="(s : in out dstring;  t : dstring)  is
begin                                  -- note that s is declared "in out"
if  s /= null  then                    -- to allow this garbage collection
                                       -- garbage collect old value of s
end if;
s := new string(t.all);
end;
```

NOTE: This proposal (in slightly different, but substantively identical form) was approved as an Ada Language Issue (LI54) by the Ada Language Issues Working Group of SIGada. The final vote, taken March 1, 1989, was unanimously in favor.

SURPRISES WITH ARRAY EXPRESSIONS

DATE: October 31, 1989

NAME: Lennart Mansson

ADDRESS: Box 4148
S-203 12
Malmo, Sweden

TELEPHONE: +46-40-25 46 39

ANSI/MIL-STD-1815A REFERENCE: 5.2, 5.2.1

PROBLEM:

Consider the following declarations:

```
subtype Short_String is String(1..3);

type T is
        record
                S-S:; Short_String;
        end record;

procedure P(S_S: Short_String);

S: String(1..5) := "adobe";

S-S: Short_String;
```

Now the following statement is allowed:

```
S_S := S(3..5);
```

But not these:

```
P(S_S => $(3..5));
T;(S_S => S(3..5));
```

This is extremely surprising to new users of the language, and I would expect that even an experienced Ada Programmer can get in this trap.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use of intermediate variable so that the implicit subtype conversion takes place when assigning to the intermediate.

POSSIBLE SOLUTIONS:

Instead of allowing subtype conversion in a special case, namely array assignment, as done in 5.2.1, it could be possible to redefine what is meant by "belongs to a subtype" for array expressions.

IMPLICIT ARRAY SUBTYPE CONVERSIONS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 5.2.1

PROBLEM:

Implicit array subtype conversion should be performed for default initialization of array-valued record components and for evaluation of array-valued components of aggregates.

For example, elaborating the following declarations should raise no exception:

```
subtype S1 is String (1 ...10);
subtype S2 is String (2 ...11);

X : S1 := "0123456789";

type R is
        record
                F: S2 := X;
        end record;

Y : R;
Z : R := R'(F => X);
```

It seems very unintuitive that the declarations of Y and Z raise an exception while a seemingly equivalent (although more awkward) formulation such as

```
subtype S1 is String (1 ...10);
subtype S2 is String (2 ...11);

X : S1 := "0123456789";

type R is
        record
                F: S2;
        end record;

Y : R;
Z : R;
begin
```

```
Y.F := X;
Z.F := X;
```

raises no exception.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use explicit conversions.

POSSIBLE SOLUTIONS:

NAMED CONSTRUCTS

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 5.3(2), 5.4(2), 9.7.1(2), 9.7.2(2)

PROBLEM:

Currently, the language allows both loops and blocks to be named. Naming these constructs achieves three purposes: (1) it allows an outer loop to be exited from within an inner loop; (2) it allows entities hidden within an inner declarative region to be referenced by selected component notation; (3) it documents the scope of the loop or the block (i.e., where it begins and ends).

The third of these purposes could be achieved for IF, CASE, and SELECT statements if these statements could also be named. For example, many style guides recommend commenting the END IF, END CASE, or END SELECT to make it easier for the reader of the program unit to find his/her way around in the code, particularly if such constructs are nested. Making such a facility a part of the language would be one further step on the way to making Ada self- documenting.

Examples:

```
has_been_found: IF found THEN
END IF has_been_found;

value_of_status: CASE status IS
END CASE value_of_status;

read_or_write: SELECT
END SELECT read_or_write;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use comments to document the end of IF, CASE, and SELECT statements.

POSSIBLE SOLUTIONS:

NEED FOR OPTIONAL SIMPLE_NAMES FOR CASE, IF AND SELECT STATEMENTS

DATE: September 14, 1989

NAME: Ray A. Robinson, Jr.

ADDRESS: SPARTA
4901 Corporate Drive
Huntsville, AL 35805

TELEPHONE: (205) 837-5282 x1472

ANSI/MIL-STD-1815A REFERENCES: 5.3(2), 5.4(2), 5.5(2), 9.7(2)

PROBLEM:

CASE, IF, and SELECT statements lack optional simple _names, which are available for loops and blocks. The structure of a program whose loops all have names is much easier to understand because the ends of each loop are connected by the loop name. Simple_names are also useful as comments, as shown in the following example:

```
test_for_file_name_too_long:
IF length . maximum_length then
     shorten_file_name;
ELSE
        null;
END if test_for_file_name_too_long;
```

IMPORTANCE: IMPORTANT

Adding optional simple_names to these statements will not only help people to write code that is easier to read, it would also improve machine-generated code. A PDL statement such as "--| Search list of candidates" would become part of the source code as simple_name "search_list_of_candidates".

Streamlined location of compilation errors is another important benefit of simple_names. A compiler could report "Missing END for test_.._long" instead of just "Missing End" followed by a host of secondary errors.

CURRENT WORKAROUNDS:

Comments can be used for the same purpose, but the accuracy of this technique depends on programmer discipline and code walk-throughs.

Another workaround is to put a named block around each structure. Unlike simple_names, this technique adds overhead that may not be acceptable in some cases.

POSSIBLE SOLUTIONS:

Add"[if simple_name:]", etc. to the definitions of CASE, IF, and SELECT statements as shown above.

Additional solution to consider: Require simple_names for IF, CASE, LOOP, block, and SELECT statements. The coding standard developed in this office already requires names for loops and blocks. This requirement used with long loop names makes a substantial improvement in program readability. One of our engineers sent a letter written on the back of scrap Ada printouts to a friend with no software or technical background; she received a reply from her amazed friend who was able to understand the Ada code on the reverse.

DECISION TABLES

DATE: August 30, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 5.4

PROBLEM:

Many common applications have complex logical conditions that factor nicely into decision tables, but Ada has no clean mechanism for defining them.

IMPORTANCE: IMPORTANT

Without such a facility, this type of application will require workarounds that are harder to read and more subject to error.

CURRENT WORKAROUNDS:

If statements with elsif clauses, each having a lengthy boolean expression.

POSSIBLE SOLUTIONS:

Changes 5.4(2) to read

```
case_statement ::=
  case list_of_expressions is
          case_statement_alternative
        {case_statement_alternative}
end case;

case_statement_alternative ::=
        when list_of_choices {|list_of_choices} => sequence_of_statements
```

list_of_choices ::= choice {,choice)

list_of_expressions ::= expression {,expression}

with corresponding changes in the text. It is an error for a list_of_choices to be longer than the list_of_expressions; if the list_of_choices is shorter, the remaining choices should be interpreted as others.

REAL CASE

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 5.4

PROBLEM:

The case statement of Ada does not allow evaluation of a real expression. In numerical computing it is common to require different processing for different ranges of a variable or calculation.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use an if with elsif clauses. This is harder to read, more error prone and hinders code optimization.

POSSIBLE SOLUTIONS:

Allow the expressions and choices for a case statement to be of arbitrary type. Allow choices to be real ranges, not just discrete ranges.

Alternatively, allow real expressions and real choices, but do not allow more general objects, e.g., structures. This would compromise regularity, but be easier to implement.

STRING CASE

DATE: September 13, 9189

NAME: Seymour Jerome Metz

DISCLAIMER: The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, VA 22003-5141

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, VA 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 5.4

PROBLEM:

The case statement of Ada does not allow evaluation of a string expression. When processing character data it is frequently necessary to execute different code for each of a set of strings.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use an if with elsif clauses. This is harder to read, more error prone and hinders code optimization.

POSSIBLE SOLUTIONS:

Allow the expressions and choices for a case statement to be of arbitrary type. In particular, allow choices to be strings, not just scalars and ranges.

Alternatively, allow scalar and string expressions and scalar and string choices, but do not allow more general objects, e.g., structures. This would compromise regularity, but be easier to implement.

CASE STATEMENT WITH CALCULATED CHOICE VALUES

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 5.4

PROBLEM:

The choice values can only be static values of a discrete type. It would be very useful if the choice could be relaxed to be any expression that returns a value of the type specified. Obviously, this would require an others alternative.

At the very least, it should be possible to have (static) strings as choice values.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Writing multibranch if statements instead. Case statements would be preferred, since they are much clearer to read.

POSSIBLE SOLUTIONS:

LOOP CONTROL

DATE: August 30, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 5.5

PROBLEM:

The current mechanism for loops cannot handle many common situations. In numerical computation, e.g., integration, it is often necessary to iterate over a range of real numbers. In symbolic computation, it is often necessary to traverse a list structure. Even when traversing a discrete range, it is often necessary to go in increments greater than one.

Frequently, conditions in the middle of the loop require that the loop be iterated.

IMPORTANCE: IMPORTANT

In the absence of more powerful loop mechanisms, clumsy workarounds will be needed, reducing efficiency, readability, and reliability.

CURRENT WORKAROUNDS:

Add initialization code before the loop and exit statements within the loop. This has the disadvantage that the semantics of the loop are scattered, instead of being concentrated in a single location, making the program harder to read and less reliable. This also hinders code optimization.

Define an additional boolean variable and put all of the loop tail code inside an if. This makes the program harder to read and less reliable. This also hinders code optimization.

Use a GOTO to a label on the END LOOP statement. This makes the program harder to read and less reliable.

POSSIBLE SOLUTIONS:

Revise 5.5(2) to specify

```
iteration_scheme ::=
  (loop_parameter_specification)
  (while condition)
(until condition)

loop_parameter_specification ::=
        identifier in [reverse] range [by expression] |
        identifier := expression next expression
```

Define a new statement:

```
iterate_statement ::= iterate [loop_name] [when condition];
```

Revise 5.7(2) to read:

```
exit_statement :: = exit [loop_or_block_name] [when condition];
```

Revise 5.7(3) to state that an exit with a block name is allowed only within the named block, and to give all relevant restrictions for exit from a block.

ADDITION OF LOOP/UNTIL CONSTRUCT

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 5.5(2)

PROBLEM:

The language provides special-case loop constructs both for test-at-the-top loops (WHILE loops) and for FOR loops, but provides no similar special-case loop construct for test-at-the-bottom loops; but provides no similar special-case loop construct for test-at-the-bottom loops; there is an annoying lack of symmetry to this.

IMPORTANCE: IMPORTANT

This revision request is motivated primarily by concerns about symmetry and aesthetics, but it also yields some practical benefit to programmers (and Ada language instructors) because a special-case test-at-the-bottom loop construct is more understandable than a loop with a test-and-exit at the bottom.

CURRENT WORKAROUNDS:

Use a loop with a test-and-exit at the bottom.

POSSIBLE SOLUTIONS:

Add LOOP/UNTIL construct to the language, similar to the "repeat/until" of Pascal. The syntax for this loop construct would be:

```
Loop_statement ::=
        test_at_bottom_loop_statement
        test_at_top_loop_statement

test_at_bottom_loop_statement  ::=
        [loop_simple_name:]
                loop
                        sequence_of_statements
```

until condition;

test_at_top_loop_statement
former loop_statement specification in LRM 5.5.2

For example:

```
loop
        Remove_Element (From_This_Buffer => The_Buffer);
until The_Buffer.Status = Empty;
```

COMPATIBILITY

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed, because a new reserved word UNTIL has been added to the language, and this word may already be in use in some code (although this is not likely).

THE LIMITATIONS TO A STEP OF ONE OF THE SPECIFICATION OF THE LOOP PARAMETER IN FOR LOOPS

DATE: October 30, 1989

NAME: Stephane Bortzmeyer

ADDRESS: INRETS
2, rue du General Malleret-Joinville
BP34
94114 Arcueil Cedex

TELEPHONE: (1) 47 40 71 07
(1) 45 47 56 06 (fax)

ANSI/MIL-STD-1815A REFERENCE: 5.5(6)

PROBLEM:

In the present standard, the loop parameter can only be incremented or decremented by a step of one. A loop with another step cannot be programmed in Ada and leads to less understandable workarounds. Or, it does not seem there is any advantage to this limitation.

IMPORTANCE: IMPORTANT

but only for a minority of programs.

CURRENT WORKAROUNDS:

With the present standard, the only way is to use a WHILE loop[1] with a parameter you declare outside of the loop. It is, for example, the solution proposed by Barnes (Programming with Ada). It has several disadvantages:

-- necessity of a declaration of the parameter outside of the loop

-- necessity to manage by yourself the incrementation

-- and, moreover, hiding of the deep nature of the loop. The WHILE loops should be reserved to the cases where you do not know in advance the number of iterations and not to emulate FOR loops.

POSSIBLE SOLUTIONS:

A solution can be to modify the definition of the language. We need to change LRM 5.5(2) in this way.

loop_parameter_specification ::=

---

[1]On a regular Loop ..END LOOP

identifier in [reverse] discrete_range
[step positive_integer_value]

For the price of the introduction of a new reserved word, step, we reach a simple solution for the problem. If the value of the step is not a submultiple of the range, we can decide that the loop will stop at the value immediately under the upper limit which is a submultiple. The step will default to 1 and, therefore, ensure compatibility with existing programs. The step will have to be positive, negative steps will be made by the reverse statement. At last, the step can also apply to non integers types, since they are indexed by integers (POS and VAL attributes, see LRM 3.5.5).

Another problem is the risk to raise an exception is the step is too large. If we want that the incrementation of the loop parameter will be made only with the SUCC attribute, without making any assumptions on the addition, we have to surround this incrementation by a block that will handle this exception. We can emulate the step loop with the following standard Ada program:

```
for I in 1 .. N step S

        { instructions }

end loop;
```

can be translated:

```
I : T;

...

I := 1;
while I <= N loop
                { instructions }
        begin
                for J in 1 .. S loop
                        I := T'SUCC (I);        --      We don't assume the existence of an
                                                        addition operator.
                end loop;
        exception
                when CONSTRAINT_ERROR => exit; -- Same as I > N.
        end;
end loop;
```

We could also avoid the raise of an exception:

```
I := 1; -- I is of type T
I_loop :
while I <= N loop
        { instructions }
        J_loop :
        for J in 1 .. S loop
                if I < T'LAST then
                                I := T'SUCC (I);
                else
```

```
                                exit I_loop;
                        end if;
                        end loop J_loop;
        end loop I_loop;
```

To prevent the loop for being unincremented, it is enough to compel the step to be strictly positive.

We can ask ourselves about the type of the increment. Its only use is in the specification of the loop J, so we can consider it as a universal integer. Any positive value can match it.

THE LIMITATIONS TO DISCRETE TYPES OF THE SPECIFICATION OF THE LOOP PARAMETER IN FOR LOOPS

DATE: October 30, 1989

NAME: Stephane Bortzmeyer

ADDRESS: INRETS
2, rue du General Malleret-Joinville
BP34
94114 Arcueil Cedex

TELEPHONE: (1) 47 40 71 07
(1) 45 47 56 06 (fax)

ANSI/MIL-STD-1815A REFERENCE: 5.5(6)

PROBLEM:

In the present standard, the loop parameter can only be a discrete type and, furthermore, it can only be incremented or decremented by a step of one. A loop with a real parameter or an integer with another step cannot be programmed in Ada and leads to less understandable workarounds. I propose elsewhere an extension of FOR loops to steps different from 1[2].

But it could be also useful to deal with real loop parameters, for example to apply a given treatment to a succession of real numbers with uniform gaps (times or intervals of measure).

IMPORTANCE: IMPORTANT

but only for a minority of programs.

CURRENT WORKAROUNDS:

With the present standard, the only way is to use a WHILE[3] loop with a parameter you declare outside the loop. It is, for example, the solution proposed by Barnes (Programming with Ada). It has several disadvantages:

-- necessity of a declaration of the parameter outside of the loop

-- necessity to manage by yourself the incrementation

-- and, moreover, hiding of the deep nature of the loop. The WHILE loops should be

---

[2]under the title: "The limitations to a step of one of the specification of the loop parameter in FOR loops"

[3]On a regular LOOP END LOOP

reserved to the cases where you do not know the number of iterations and not to emulate FOR loops.

POSSIBLE SOLUTIONS:

A solution can be to modify the definition of the language. The solution I propose here is very close from the one I submitted for the discrete types, introducing a new reserved word, step.

But for the real loop parameters, the situation is a little more complicated. A floating-point parameter is out of question[4], because the precision will be difficult to foresee and will vary in the interval. On the contrary, fixed-points parameters seems possible in this way:

```
loop_parameter_specification ::=
        identifier in [reverse] discrete_range
                [step positive_integer_value] |
        identifier in [reverse] fixed-point_interval
                [step real_value]

fixed-point_interval ::=            see 3.5.9(2))
        fixed_point_constraint |
        fixed-point_subtype_definition
```

The real value of the step should obviously be of the same type as the parameter or be an universal fix.

At the contrary of the discrete types which have the attributes which have the attribute SUCC, we have to assume the existence of the addition operator, possibly redefined by the programmer. The following program:

```
for I in 1.0 .. N step S
        -- where S is from a fixed-point
         { instructions }
end loop;
```

can be translated:

```
I := 1.);
while I <= N loop
        { instructions }
        begin
                I := I + S;
        exception
                when CONSTRAINT_ERROR => exit; -- Same as I > n.
        end;
end loop;
```

[4]Except if the step is much larger than the interval between two model numbers. As the interval varies during the loop you can see the difficulty.

FOR LOOP DOES NOT BECOME COMPLETED

DATE: August 27, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne, IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 5.5(5,8)

PROBLEM:

In a for loop, the execution of the loop statement is complete if, when evaluated, the discrete range is a null range [5.5(80]. Otherwise the sequence of statements making up the loop is executed once for each value of the discrete range; however the LRM does not indicate that the for loop statement becomes completed when the discrete range is exhausted.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: N/A

POSSIBLE SOLUTIONS:

Add the following type sentences to the end of 5.5(8). "When all values of the discrete range have been assigned to the loop parameter and execution of the sequence of statements has occurred once for each value then execution of the loop statement is complete. If the loop is left as a consequence of the execution of an exit statement or some other transfer of control, then execution of the loop statement is complete.

EXIT STATEMENT TO COMPLETE EXECUTION OF BLOCK STATEMENT

DATE: October 20, 1989

NAME: Thomas J. Quiggle

ADDRESS: Telesoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (619) 457-2700 ex. 158

ANSI/MIL-STD-1815A REFERENCE: 5.6 and 5.7

PROBLEM:

Loop statements, functions, procedures, and entries can all be completed via a single nested statement. Block statements lack such a capability. Program clarity could be greatly improved if such a capability were provided. The following example illustrates this point.

```
BLOCK: declare
        ...
begin
        if SOME_CONDITION then
                ...
                if ANOTHER_CONDITION then
                        ...
                        if YET_ANOTHER_CONDITION then
                        ...
                        -- No additional code (points A and B below)
                        -- need to be executed under these
                        -- conditions.  Useful execution of the block
                        --is complete.
                        ???
                end if;
                ...
         end if;
         ...        --(A)
  end if;
  ...          --(B)
end BLOCK;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

In the absence of an exit mechanism for a block statement, three workarounds come to mind: a goto to the end of the block, exiting a "trivial loop" surrounding the block, or exiting the block via an exception. All

three obfuscate the intended execution of the block. The last two are likely to introduce unnecessary overhead at run time. Examples of these three mechanisms follow:

Example 1 - Goto to the end of the block:

```
BLOCK: declare
        ...
begin
        ...
        goto BLOCK_END;
        ...
        <<BLOCK_END>> null;
end BLOCK;
```

Example 2 - Exit enclosing loop:

```
for INDEX in 1..1 loop
BLOCK: declare
        ...
begin
        ...
        exit;-- Enclosing loop
        ...
end BLOCK; end loop;
```

Example 3 - Exit via exception:

```
BLOCK: declare
        GET_ME_OUT_OF_HERE : exception;
begin
        ...
        raise GET_ME_OUT_OF_HERE;
        ...
exception
        when GET_ME_OUT_OF_HERE => null;
        when others => raise;
end BLOCK;
```

POSSIBLE SOLUTIONS:

In section 5.7, extend the definition of an exit statement declaration to read:

```
exit_statement ::= loop_exit_statement | block_exit_statement

loop_exit_statement ::=
        exit [loop_name] [when condition];

block_exit_statement ::=
        exit block_simple_name;
```

In order to provide upward compatibility for existing Ada applications, the exit statement used to complete

a block would require a block_simple_name. In the following code fragment:

```
loop
        ...
        BLOCK: begin
                ...
                exit; -- Legal Ada83 syntax. Completes execution of enclosing loop
                ...
        end BLOCK;
        ...
end loop;
```

the exit statement would continue to complete the (unnamed) loop statement, rather than the nested block.

COMMON PROCESSING TO EXCEPTION HANDLERS OF THE SAME FRAME

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 5.6(2), 6.3(2), 9.1(3), 7.1(2)

PROBLEM:

Many times there is common processing among exception handlers of the same frame. There is not mechanism to allow this.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

No workarounds exist.

POSSIBLE SOLUTIONS:

Change:

```
begin
        sequence_of_statements
[exception

   exception_handler
   {exception_handler}]
end
```

to:

```
begin
        sequence_of_statements
[exception
        [sequence_of_statements]
   exception_handler
   {exception_handler}]
end
```

RAISEWHEN

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 5.7, 11.3

PROBLEM:

Ada has a restricted raise statement which does not have the when option of an exit statement:

```
exit [loopname] [when condition] ;
raise [exception_name];
```

The raise statement syntax should be changed to:

```
raise [exception_name] [when condition] ;
```

in order to parallel the exit statement, since they both do fundamentally similar things, namely exit from some environment under certain conditions.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

if condition **then raise** exception_name; **end if;**

POSSIBLE SOLUTIONS:

add a when option to the raise statement as described above.

The exit statement has a when option, even though, strictly speaking it does not need one. The following are equivalent:

```
exit Loop_name when condition;
if condition then exit loop_name; end it;
```

However, the designers of Ada put the former into the language for the sake of convenience. The raise statement is even more likely to be used as a special case. as in the following condition:

```
procedure push ( Item: in thing; Onto: in out Stack ) is
begin
        raise Stack_overflow when is_full( Stack );
        Onto.length := Onto.length + 1;
        Onto.Value( Onto.length ) := Item;
end push;
```

Some languages have explicit capabilities of providing preconditions and postconditions in programs (sometimes collectively referred to as assertions) by means of a special statement. For example, the raise statement above might be replaced in some languages by:

```
assert not is_full ( stack );
precondition( not is_full( stack ));
```

Ada does not need this because of the raise statement, but it would clean up the syntax a little.

The other reason to add it is that the word raise is then the first word on the line and more likely to catch the eye of the reader. In the statement:

```
if is_full( stack ) then raise stack_overflow; end if;
```

the word raise is buried partway across the line and less likely to be seen as the major signpost that it is.

EXIT AND RETURN IN A LOOP

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 5.7

PROBLEM:

Exit and return in a loop.

Software engineering purists impose a rule: do not use exit or return statements, they make programs harder to understand.

Well, what exactly is the problem? Let's inspect the two alternatives:

Ada allows writing:

```
while SOME_CONDITION
loop
...
        exit when OTHER_CONDITION
...
end loop;
```

Software engineering purists require rewrite to something like:

```
declare
        BUSY:BOOLEAN :=TRUE;
while SOME_CONDITION and BUSY
loop
...
        if OTHER_CONDITION
        then BUSY :=FALSE;
        else...
        end if
end loop;
```

Clearly the rewritten version has some disadvantages:

* introduction of a declaration

* BUSY := FALSE; is a little more hidden than the "exit when"

* BUSY could be modified in a little less straightforward manner, e.g. passing BUSY as an in out parameter to some subprogram.

* exit will draw more attention than BUSY := FALSE if one uses a text representation that shows reserved words in boldface, as suggested by the reference manual.

* the two sequences of statements are no longer on the same syntactic level

But there is one clear advantage: when one examines the while loop_iteration_scheme, one knows that in order to understand the loop, one needs to watch out for BUSY := FALSE statements. Apart from this, finding and understanding the BUSY modifications will be at least as difficult as finding the "exit when" statement (a little more in fact since the "exit when..." construct was specially invented because its equivalent "if.. then exit end if" draws less attraction)

The same problem appears even stronger when using the for iteration_scheme. There are plenty of software engineering aspects about for loops with respect to their while alternative. One doesn't need a separate declaration that repeats the required range, the loop variable is protected inside the loop,etc... One would be tempted to say that a loop tells you the number of iterations and the value of the loop variable for each iteration. Again this is not true when using exits. In that case the software engineering compromise is even harder to make: either one stays "pure" and one forbids the exit, or one loses the guarantee that the number of iterations is unpredictable. The first approach requires that those loops that logically imply an exit are turned into while loops, and hence they lose almost all software engineering advantages of the for loop, where the second alternative loses the predictability of the number of iterations, but on the other hand it keeps all other advantages of the for loop: one knows the sequence of values of an automatically declared and protected loop variable until execution terminates by an exit.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: House rules?

POSSIBLE SOLUTIONS:

One would have the best of both worlds if one would be able to define in the loop iteration scheme whether exits to the end loop are allowed or not. The hard part of this approach is once more the decision not to add reserved words to the language, otherwise the following code is "ideal" with respect to the problem perceived by the purist:

```
while SOME_CONDITION
loop exitable
...
        exit when OTHER_CONDITION
...
end loop;
```

There are two problems with this approach, due only to the fact that Ada9X needs to be full upward compatible. A decision that probably severely hamper Ada's evolution further towards a software engineering based language, but still a decision made. One is that one cannot introduce a new keyword, the other is that all current non-purist Ada code use exits form loops without announcing it. Surely, from

a software engineering standpoint it is better to announce unsafety than it is to announce purism, but this amount to a Ada83 compatibility problem;

So, why not consider:

```
for I in L .. U
limited loop
...
        exit when OTHER_CONDITION; --compilation error
...
end loop;
```

This solution does not introduce a new keyword (still overloads on though), and the software engineering principle could be regained if Ada reformatters would be able to add the "limited" keyword before any loop that does not contain an exit.

A last approach, with almost no impact on the language would be to allow loop exits only when they use the syntax "exit loop_name". This would clearly show the level of exit taken and announce its possibility immediately before the loop iteration scheme. But of course, there is no way this could be introduced in the language without affecting full upward compatibility. This approach could solve the problem perceived only when fully supported by an APSE syntactical editor.

EXIT FROM BLOCK

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 5.7

PROBLEM:

Ada has one named construct that does not have an exit or return that should. That construct is the block statement.

IMPORTANCE: ADMINISTRATIVE (for consistency)

CURRENT WORKAROUNDS:

Use a "goto" statement to a label placed either just before or just after the end of the block.

POSSIBLE SOLUTIONS:

Allow an exit statement to exit the construct in the form:

```
exit Block_Name;
```

The name should be required both on the block and in the exit statement when this feature is used to prevent conflict or confusion with its use under the Ada83 standard.

EXITING BLOCKS

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 5.7

PROBLEM:

It should be possible to exit (named) blocks.

Extending the functionality of the exit statement in this way would be beneficial to code legibility.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

use nested conditional statements.

POSSIBLE SOLUTIONS:

Allow exits from (named) blocks.

WHEN/EXIT CONSTRUCT

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 5.7(2)

PROBLEM:

The syntactic order of the EXIT/WHEN construct is backwards and is inconsistent with the IF/THEN construct in particular and the English language in general: the condition should be evaluated before the action (e.g. exiting) is specified.

IMPORTANCE: IMPORTANT

Students have been known to complain that they have to invert their thinking to make the syntax of the EXIT/WHEN construct work correctly: any such obstacle (no matter how trivial) is yet another disincentive to learning the language.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Invert the syntax of the EXIT/WHEN construct as follows:

```
exit_statement :=
        [when condition] exit [loop_name];
```

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed (although it is easy to fix).

ELIMINATION OF RETURN STATEMENT EXCEPT IN FUNCTIONS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 5.8

PROBLEM:

The use of the RETURN statement to return from procedures encourages bad programming habits. This is only slightly less horrible than using a goto or an exception to control normal execution. Code written with many returns from procedures is incredibly difficult to understand, because the flow-of-control is obscured.

IMPORTANCE: IMPORTANT

Returns from procedures should never have been allowed in the standard in the first place.

CURRENT WORKAROUNDS:

Good programming discipline, coding standards, code audits, code checking utilities.

POSSIBLE SOLUTIONS:

Eliminate the use of RETURN statements except inside functions.

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed. On the other hand, only badly-written code will fail to compile, so this is not great loss.

WHEN WITH RETURN AND RAISE

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 5.8(2), 11.3(2)

PROBLEM:

Currently, the language allows a WHEN construct with an EXIT statement as a shorthand equivalent to placing the exit statement within an IF statement. Since, however, the EXIT statement is similar in function to the RETURN and RAISE statements (all three are controlled GOTO's), it would be convenient to be able to use a WHEN construct with these other two types of statement as well.

Examples:

```
RETURN WHEN ptr = NULL;
RETURN true WHEN found;

RAISE WHEN value > maximum;
RAISE out_of_bounds WHEN index NOT IN list'RANGE;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use an IF statement.

POSSIBLE SOLUTIONS:

For additional references to Section 5. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

NUMBER	TITLE	PAGE
0141	INCLUDE "WHEN" IN RAISE STATEMENT SYNTAX	11-16
0310	BLANK PADDING FOR STRING ASSIGNMENTS	4-55
0413	ATTRIBUTES AS FUNCTIONS	16-104
0504	PROPOSAL FOR AN EXCHANGE OPERATOR	2-22
0609	REDEFINITION OF ASSIGNMENT AND EQUALITY OPERATORS	4-78
0734	INCONSISTENT TREATMENT OF ARRAY CONSTRAINT CHECKING	4-50
0745	INTELLIGENT STRONG TYPING	3-67

```
private
-- Full type declaration instead of incomplete just to show the purpose.
        type T_PERSON_OBJ is
                        record
                                FIRST_CAR : T_CAR;
                                ...--person specific components
                        end record;

        type T_CAR_OBJ is
                        record
                                OWNER : T_PERSON;
                                NEXT_CAR_OF_SAME_OWNER : T_CAR;
                                ...-- car specific components
                        end record;

        type T_PERSON is access T_PERSON_OBJ;
        type T_CAR is access T_CAR_OBJ;
end PERSONS_HAVE_CARS;
```

This practice of pointer reversal for implementing 1-to-N relationships is a quite popular and efficient technique to implement the operations supported in the example (double linked lists are even more popular but let's simplify). Similar data structures exist for

The actual problem is that we have here a set of conflicting software engineering principles:

* One should model the universe of a software project as a set of types linked by a set of relations. Or for the purists, one should view the universe as a set of unary and multi-ary relations.

* One should obtain efficient computer support for this view on the universe.

* One should separate types and their support in distinct packages, split the support in declaration and body, and hide all implementation details in the body.

Now, as far as Ada is concerned, efficient support in the sense of pointer inversion means that types cannot be declared in separate packages. The question one should dare to ask oneself is whether this is really a fundamental contradiction or whether it is the result of a design option. The POSSIBLE SOLUTION section will try to demonstrate that a minor full upward compatible addition is sufficient to solve the problem.

As said before, I already heard a lot of surprising workarounds that just can't work. One is to use a list of dummy objects that reference CARS, this clearly doesn't work since one can never find the owner of a car without visiting all persons. Another is to use a generic package PERSON that has a formal type OWNED_OBJECT and generic CAR that has a formal type OWNER, which doesn't work either since one needs to instantiate PERSON with the actual CAR and CAR with the actual PERSON. At last, there are those who are convinced that separation of specifications and body is powerful enough to break any loops, but clearly this is not true for loops in specifications.

IMPORTANCE: IMPORTANT.

CURRENT WORKAROUNDS:

Use either less efficient, less modular or less clean implementations.

Less efficient by using an associative data structure (e.g. a tree structure or hash table) that associates cars with owners.

Less modular by merging packages as shown above. If one takes this option consistently, one ends up with a single package.

Less clean by allowing CARS to store a general purpose value of type SYSTEM.ADDRESS, this way allowing the package body PERSON to implement the reverse pointers by type conversion.

POSSIBLE SOLUTIONS:

A first, primitive solution would be to allow referencing private types even before they are defined. The restriction of direct recursion (without access types) of type declarations stays as it currently is : compilers need to check this. To be able to reference these private types, one needs to remove the restriction of loop-free with clauses for specifications (the restriction may remain for bodies). It also means that compilers need to delay their code generation step until all types are defined, and this moment is no longer immediately after the compilation of specification. Since most compilers retain already the parse tree for debugging or viewing purposes, this is not so hard to implement. In short, this solution extends the private types to the notion of the incomplete types across packages.

The result of this would look like:

```
with CARS;
package PERSONS is
        type T is limited private;

        -- Support for Persons
        procedure CREATE_PERSON (P : in out T_PERSONS; NAME...);
        function NAME (P_PERSON...

        -- Support for Cars
        procedure CREATE_CAR (C : in out T_CARS; COLOR...);
        procedure COLOR (C_CAR...

        -- Support for the 1 to many relationship between PERSONS and CARS

                procedure TRANSFER_CAR_TO_PERSON (C: CARS.T; P : PERSONS.T);
        --Makes P the new owner of C

        function HAS_OWNER (C:CARS.T) return BOOLEAN;
        procedure GET_OWNER (C:CARS.T; P: in out PERSONS.T);

        generic
                with procedure FOR_EACH (C: CARS.T);
        procedure FOR_ALL_CARS_OF (P:PERSONS.T);
        -- Performs FOR_EACH for each car of P.

        private
                type T_OBJ;
```

```
        type T is access T_OBJ;
end PERSONS:

with PERSONS;
package CARS is

        type T is limited private;

        procedure CREATE_CAR (C : in out T_CARS; COLOR...);
        procedure COLOR (C_CARS.T...

-- Support for pointer reversal operations for the body of PERSONS.
-- Simplified but sufficient interface.
package POINTER_REVERSAL_FOR_PERSON is
        procedure SET_OWNER (C: CARS.T; P : in PERSONS.T);
        procedure GET_OWNER (C: CARS.T; P : out PERSONS.T);
        procedure SET_NEXT (C: CARS.T;NEXT_CAR: in CARS.T);
        procedure GET_NEXT (C: CARS.T;NEXT_CAR: out CARS.T);
end POINTER_REVERSAL_FOR_PERSON;

private
        type T_OBJ;
        type T is access T_OBJ;
end CARS;
```

Another, more software-engineering minded solution would be to allow additional specifications for a single package. This feature should not be confused with two separate packages: the feature is special in that sense that the additional specification is considered part of the same declarative region as the incomplete specification (and its body).

This feature is not only something which solves the problem described before, but also comes in handy in many other occasions where one designs packages with a two-level interface. E.g. "agent" packages that control communication between end-users and general services. Such packages currently have to specify their interface to the end-user in the same compilation unit as their interface to the general services, while the latter interface is typically of no interest to the end-users.

The best and certainly the most consistent way to model this solution would be to extend the principle of body stubs to package specifications. Note that this notation does not introduce any ambiguity, it adds two new declarations:

```
package_stub ::= package package_simple_name is separate;
subpackage_declaration ::= separate (parent_unit_name)
        package_specification;
```

which can be easily distinguished by from body_stub and subunit by the absence of the reserved word body.

This solution avoids the need to allow cycles in with clauses, since the recursion is now hidden by the split package:

```
package CARS is
```

```
        type T is limited private;

        procedure CREATE_CAR (C : in out T_CARS; COLOR...);
        procedure COLOR (C_CARS.T...

        package POINTER_REVERSAL_FOR_PERSON is

private
        type T_OBJ;
        type T is access T_OBJ;
end CARS;

with CARS;
package PERSONS is
        type T is limited private;

        -- Support for Persons
        procedure CREATE_PERSON (P : in out T_PERSONS; NAME...);
        function NAME (P_PERSON...

        -- Support for Cars
        procedure CREATE_CAR (C : in out T_CARS; COLOR...);
        procedure COLOR (C_CAR...

        -- Support for the 1 to many relationship between PERSONS and CARS

                procedure TRANSFER_CAR_TO_PERSON (C: CARS.T; P : PERSONS.T);
        --Makes P the new owner of C

        function HAS_OWNER (C:CARS.T) return BOOLEAN;
        procedure GET_OWNER (C:CARS.T; P: in out PERSONS.T);

        generic
                with procedure FOR_EACH (C: CARS.T);
        procedure FOR_ALL_CARS_OF (P:PERSONS.T);
        -- Performs FOR_EACH for each car of P.

        private
                type T_OBJ;
                type T is access T_OBJ;
        end PERSONS:

        with PERSONS;
        separate (CARS)
        package POINTER_REVERSAL_FOR_PERSON is
                procedure SET_OWNER (C: CARS.T; P : in PERSONS.T);
                procedure GET_OWNER (C: CARS.T; P : out PERSONS.T);
                procedure SET_NEXT (C: CARS.T;NEXT_CAR: in CARS.T);
                procedure GET_NEXT (C: CARS.T;NEXT_CAR: out CARS.T);

end POINTER_REVERSAL_FOR_PERSON;
```

Purists may prefer an additional with clause that references such separate packages, such that POINTER_REVERSAL_FOR_PERSON is not automatically visible to the packages that don't with it explicitly. This will need introduction of name instead of a simple_name in the with clause, and might be considered less consistent with the body stubs.

Maybe a last remark : negative impact on the language is none for the practical minded user (don't use it if you don't like it, exactly as with body stubs). Positive impact on the language is substantial as far as practical modular design will no longer he hampered by the problems perceived above.

PARAMETER MODES WITH ACCESS TYPES

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 3.8(5)

PROBLEM:

The Reference Manual states that if an access object is a constant, only the access value cannot be changed; in particular, the designated object can be. [3.8(5)] This has consequences when a formal parameter is of an access type. Making the parameter mode IN provides no insurance that the designated object will not be changed inside the subprogram, task, or generic unit. Conversely, making the parameter mode OUT does not insure that the designated object cannot be read inside the subprogram or task.

While this may seem to be technically correct, it causes serious practical difficulties. Most of the time, a programmer who declares a type to be an access type thinks of the type in terms of its designated type, not as an access type. It would seem reasonable to allow the access type to behave the same way as its designated type, especially since notationally an access object behaves as if it were really its designated object (e.g., pointer.component behaves like a record, pointer(3) behaves like an array).

A more serious problem concerns the use of private types. One of the purposes of using a private type is to protect code external to a package from changing if the implementation of the type were to change. But in this case, if a type were changed from, say, a record to a pointer to a record, the meaning of the parameter modes (except for IN OUT) would no longer be the same. IN would no longer mean that the entity represented by the type cannot be updated; OUT would no longer mean that the entity represented by the type cannot be read. The altered meaning of the parameter modes is an unintended side effect of making the private type an access type.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Specify the parameter mode as if the parameter were of the designated type.

POSSIBLE SOLUTIONS:

NON-CONTIGUOUS ARRAYS

DATE: December 18, 1988

NAME: Ken Garlington (General Dynamics, JIAWG)

ADDRESS: General Dynamics
Data Systems Division
P.O. Box 748 (MZ 5997)
Fort Worth, TX 76101

TELEPHONE: (817) 762-9204

ANSI/MIL-STD-1815A REFERENCE: 3.8(8)

PROBLEM:

SUMMARY

Some embedded systems have constraints imposed on particular data structures due to the requirements of interfacing with external hardware devices. One of the problems included in this domain is the declaration of data structures which logically are arrays, but whose elements (due to the hardware interface) are non-contiguous. There does not appear to be an adequate way of declaring such an object in Ada.

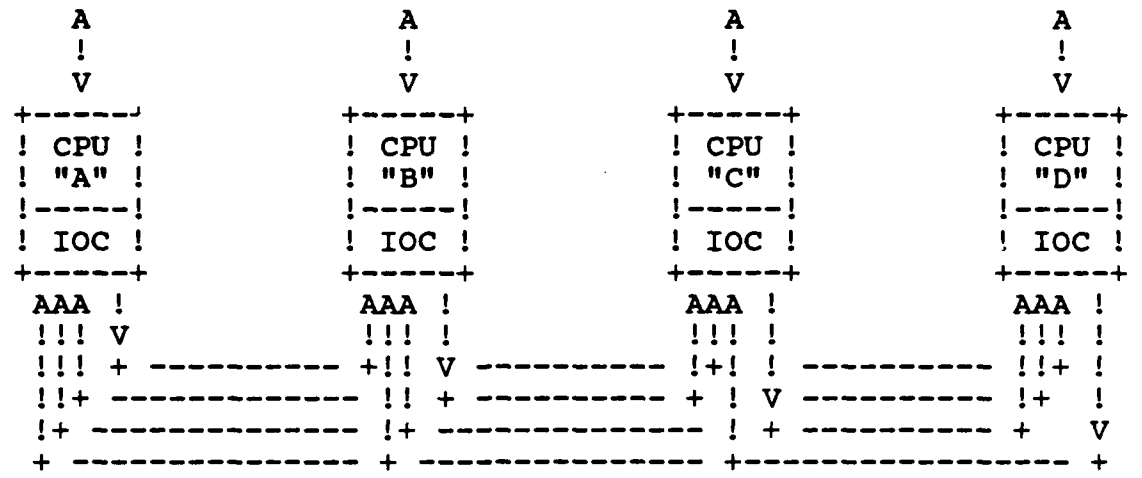
SPECIFIC REQUIREMENT:

Most of the ground rules for a viable solution are given in a related Ada change request, "Static Ragged Arrays" (see References section), and will not be repeated here.

JUSTIFICATION/EXAMPLES/WORKAROUNDS:

1. Introduction: Redundant Systems Architecture

Certain embedded systems, such as flight control or propulsion control systems, are required to exhibit high reliability due to the loss of life or property which would result from an inability of the system to continue to function. One architecture which is commonly used in these systems is a physically-redundant processor set using uni-directional communications channels ("data links"). These "data links" are used to share input values received by each processor as well as outputs generated within each CPU. These redundant values are then processed by specialized CPU algorithms called "voting planes" or "selector/monitors." An architecture diagram for a system with four processors would look as follows:



An asynchronous microcontroller, called an Input/Output Controller or IOC, performs a direct memory access (DMA) read of a pre-defined block of the CPU's memory and transmits the data within the block to the other CPUs (also called "branches" or "channels", of the system. The system as a whole also interacts with other systems (such as pilot display processors) via a redundant system I/O device such as a MIL-STD-1553 communications bus. Examples of such systems which follow this basic architecture (as either a triplex or quadruplex processor architecture) include the digital flight controls for the F-15, F-16, F-18, and F-111 aircraft, as well as digital engine control systems for various military aircraft.

2. IOC Data Blocks

>From the standpoint of a particular processor, there are four pre-defined I/O memory blocks (one transmit and three receive blocks) in a quad-redundant, which might be defined as shown below:

Addr	Contents
C000	"self" block (transmit area) -- analog signals hard-wired into branch (INTEGERs) -- hard-wired discrete signals (packed arrays of BOOLEAN) -- data received from other systems (via inter-system bus) -- processor outputs
C400	"right" block (receive area) -- copy of "self" block as maintained by another branch
C800	"opposite" block (receive area) -- copy of "self" block as maintained by a third branch
CC00	"left" block (receive area)

```
        -- copy of "self" block as maintained by a fourth branch
```

Although the four IOC blocks are shown as consecutive 1K areas of memory, they are not necessarily consecutive for every implementation. The internal arrangement of each block is decided by the pre-defined interface with the IOC unit and must be matched in order to properly process "hard-wired" analog and discrete signals. Spare locations ("holes") will be scattered throughout the block, and represent unused capability in the hardware interface.

The definitions of "self", "left", "right" and "opposite" are static (known at compile-time). The relationship of these four blocks to the data contained in CPUs "A", "B", "C", and "D" is not known until the program is executed, at which time it reads the identification of "self" (either A, B, C, or D, as determined by two discretes which are set to different values for each CPU), and maps the other branches according to the following table:

if "self" is:	other branches are:
A	left: D right: B opposite: C
B	left: A right: C opposite: D
C	left: B right: D opposite: A
D	left: C right: A opposite: B

Each of these blocks is traditionally mapped to a record structure in Ada, since there are many different types of data within the block. Furthermore, the block's contents may vary due to slightly different interfaces between each CPU and its external environment. Therefore, the record is given a discriminant, and different variants of the record describe the differences between CPUs. (Since the discriminant's value is not known until run-time, and since the mapping of the block to memory is static, the discriminant is usually ignored in the code via the pragma SUPPRESS(DISCRIMINANT_CHECK).)

2.1 Data Monitoring Functions

Each processor contains four copies of a redundant input datum: one locally-received copy in the self block (which is then transmitted to the other three branches), and one copy from each of the other three branches' self block. The values of the copies for a particular data element (i.e., control stick position) may vary between branches, either due to a data path failure or due to tolerance variations for that branch's I/O hardware. A "monitoring" algorithm is used to detect inter-branch differences, isolate the erroneous copy and reject that value for subsequent processing. In addition, the error is usually reported to the pilot if it reduces the safety or capability of the aircraft.

Although there are different algorithms for analog and discrete data types, the same algorithm is usually executed for a set of redundant values within a single subprogram. For example, a discrete monitor will perform a "majority vote" of three non-failed copies of a discrete word and reject a copy which does not match the other two. (A simultaneous dual or triple fail, although it would cause the monitor to be unable to isolate the failure, is usually so rare that it can be neglected.) To avoid declaring "nuisance" failures, most monitors require a failure to persist for a given amount of time or (for analog signals) for the differences in the copies to exceed some threshold limit which represents the allowable variations in the hardware electronics.

There are three problems associated with such an algorithm:

1) Since the monitor does identical processing on a series of data items, it is convenient to treat the input to the algorithm as three arrays of data. However, the locations of the data are fixed (in the

data block) and may not be contiguous within the block (due to the spare words, etc.).

2) The three sets of values to be used in each branch must be identical to obtain a consistent reporting of results. Usually, when there are no failures in the system, the data from branches "A", "B", and "C" are used. However, the locations of these branches' data varies depending upon the branch. Branch "A" will use the self, right and opposite blocks, while branch "B" will use the left, self and right blocks.

3) Depending upon the system, a great deal of data may need to be processed by the monitor. Therefore, the processing time (and, to a lesser extent, the memory) used must be minimized to avoid overloading the system. Such a consideration usually prohibits implementing CASE statements or similar constructs to handle the inconsistencies introduced by the previous two problems during the normal monitoring sequence (although they can be used when a failure is detected, since failures are not considered part of normal functioning).

Thus, what is needed are three arrays representing values of branches "A", "B" and "C" (and also "D", since it is used in the monitoring process after a first failure), where the components are non-contiguous, allocated statically, and are associated with a particular array at run-time.

2.2 Data Selection Functions

In order to have consistent processor outputs, each CPU must use the same inputs. (Note that values of copies may differ without being marked as failed, either because they do not differ long enough or because their differences do not exceed the threshold.) Therefore, a selector or "voting plane" is used. Typically, a selector picks a value which is in the majority (for discrete values) or in the middle (for analog values) of a set of three values. The same problems given above for the monitoring algorithm also apply to the selection algorithm.

IMPORTANCE:

See Static Ragged Arrays 9X-00018/00, 88-10-23 non-suppo. impact. The problems associated with this workaround are described fully in "Ragged Arrays" and will not be repeated here.

It should be noted that this problem is expected to exist on future redundant systems as well, given that the DMA buffer design approach has significant advantages in terms of the reliability of the system.

CURRENT WORKAROUNDS:

See Static Ragged Arrays 9X-00018/00 88-10-23

As described in "Ragged Arrays", the concept of static pointers is also used for input selection and monitoring. For a quadruplex system, four arrays of static pointers are declared and initialized to point to redundant copies of branches A, B, and C's values on power-up. The algorithms then execute periodically without any overhead required for "decoding" the positions of the inputs. The selected output value is scaled and converted to a floating-point value (if necessary) and stored into a local copy outside the IOC areas. After a first failure, the failing array component is set to point to branch D's copy and the selector/monitor continues (again, without extra overhead).

POSSIBLE SOLUTIONS:

See "Ragged Arrays."

OTHER INFORMATION:

Difficulties to be considered:

See "Ragged Arrays."

references/supporting material:
Garlington, K.E. Ada 9X Change Request No. 2A, "Static Ragged Arrays." A9X-00018/00, October 23, 1988.

RELAX DECLARATION ORDER RESTRICTIONS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 3.9

PROBLEM:

The distinction between basic declarative items and later declarative items should be eliminated. For example, the following should be legal:

```
declare
        function Foo return Integer is
        begin
                        return 17;
        end Foo;
        X : Integer := Foo;
begin
        null;
end;
```

It does make sense to disallow program unit bodies (and stubs thereof) between the introduction of a private or incomplete type and its completion, but this should be a semantic check. The present syntactic rule is too restrictive.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

One can usually work around this problem by introducing a bodyless package, as in

```
declare
        function Foo return Integer is
        begin
                        return 17;
        end Foo;

        package Skin is
                        X : Integer := Foo;
        end Skin;
        use Skin;
```

```
begin
        null;
end;
```

but nonetheless it is annoying to have to do this (and there do exist obscure cases where this is not a semantics-preserving transformation; for example, wrapping a package skin around a task variable declaration can cause a different exception handler to be selected if the task's activation raises Tasking_Error).

POSSIBLE SOLUTIONS:

GENERIC INSTANTIATIONS AS BODIES

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 3.9

PROBLEM:

It is a reasonable and customary practice to write a subprogram specification before designing the body. The specification is not intended to indicate the manner of implementation, but only to indicate the interface, and, by comments, the effect. When an exported subprogram is provided by instantiation of a generic subprogram, this separation is impossible, since an instantiation is not a body.

The user of the subprogram should not care that it is provided by instantiation, and thus should not see this fact in the specification. In writing a package, one wants to write in the specification:

procedure Decrement(X: temperatures);

and then decide later in the implementation of the body that Decrement should be an instance of a generic:

procedure Decrement **is new** operations.lessen(by => 2.5);

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Instantiate the subprogram in the specification or instantiate it in the body with a pseudonym and write a body which only calls the instantiation, using pragmas inline. The first alternative violates information protection. The second introduces an extraneous name and rely for efficiency on a pragma which may be ignored by the compiler.

POSSIBLE SOLUTIONS:

Permit a generic instantiation to be a body.

MANY DESCRIPTIONS IN THE REFERENCE MANUAL NEED TO BE CLARIFIED

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 3.9 et al

PROBLEM:

Consider the definition of 'declarative part', page 3-43. "A declarative part contains declarative items (possibly none)." That definition contains no information, and certainly provides the reader no indication of any special situations that may be lurking. And that's the entire definition; the rest of the section deals with syntax and the effect of elaboration.

Well, okay, there are several places in which a sequence of declarative items can occur. Subprograms, tasks, block statements, package specifications, package bodies come to mind immediately. But wait a moment, those declarations in a package specification are not a declarative part? That's right, even though that section of a package specification contains "declarative items (possibly none)", it is not a declarative part, see page 7-1. (Those other constructs mentioned do in fact contain declarative parts.) Presumably there's a difference worth knowing between what can go into a package specification's declarations and a declarative part.

A package spec's declarations consists of some number of 'basic declarative items' (page 7-1). A declarative part consists of some number of 'basic declarative items' followed by some number of 'later declarative items'. So it looks like the difference between the two is that you can have 'later declarative items' in a declarative part but not in a package spec's declarations. Even though that is what the language says, you'd be wrong to say that and drop the matter. The 'later declarative items' class, page 3-43, is built from the following: 'body' or 'subprogram declaration' or 'task declaration' and several other alternatives. But we're quite used to having subprogram declarations in package specifications even though it is an element of a class that cannot appear there. Interesting. In order for subprogram declarations to appear in package specifications then, subprogram declarations have to be both 'basic' and 'later' declarative items. At this point, we'd be interested in an explanation of the difference between 'basic' and 'later' declarative items; but we look in vain for any such explanation. We do note that 'basic declarative items', are also defined on page 3-43 to be: 'basic declaration' or 'representation clause' or 'use clause'. So we look in the references at the bottom of the page for 'basic declaration' and find no entry. Why not? Most other constructs used in the section seem to be there. The index guides us to page 3-1, where we find, not at all surprisingly, that a subprogram declaration is a 'basic declaration', which makes it also a 'basic declarative item'. Thus a subprogram declaration is both a 'basic' and a 'later' declarative item. It turns out that every 'later declarative item' except 'body' is also a 'basic declarative item'.

So now we know the difference between what can go in a package spec's declarations and a declarative part, right? The package spec cannot have bodies, while a declarative part can. Well, no, that's not completely right yet, either. We recall that we've gotten used to the idea of declaring private types in package specs.

But the description of package specs on pages 7-1, 7-2 and the top of 7-3 does not contain any mention of private type declarations (which is different from the private part of a package specification). The syntax says we can utilize 'basic declarative items' in that region. 'Basic declarative items', as we've seen are rep clauses, use clauses and basic declarations. So we check basic declarations again and find there is no way to get a private type declaration out of the basic declarations. This is troubling. The description of package specs doesn't reference private type declarations. However, the section on declarative part does reference the section on private type declarations. This is curious since private type declarations may not appear in declarative parts. I can't think of any good reason why private type declarations are referenced where they can't be used and not referenced where they can.

So we go to page 7-5 and find the sentence "A private type declaration is only allowed as a declarative item of the visible part of a package, or ...". This looks like an unreferenced extension of the material in section 7.1 to me. 7.1 contains no indication that 'basic declarative items' in that context are intended to include private type declarations. Yet some pages down the line the extension is made. This all is unnecessarily complicated. And it seems especially confusing that whoever developed section 7.1 of the LRM went to so much trouble to exclude bodies from the syntax for package specs but did not at the same time include a reference for private type declarations.

This marginally mystical tour through a small part of the LRM came about as I was checking the accuracy of some material for an introductory Ada class. The particular paragraph that launched me is 9.3:2.

"If an object declaration that declares a task object occurs immediately within a declarative part, then the activation of the task object starts after the elaboration of the declarative part (that is, after passing the reserved word begin following the declarative part); similarly if such a declaration occurs immediately within a package specification, the activation starts after the elaboration of the declarative part of the package body. The same holds for the activation of a task object that is a subcomponent of an object declared immediately within a declarative part of a package specification. The first statement following the declarative part is executed only after conclusion of the activation of these task objects."

Huh? What does that mean? It turns out that after you wade through all those words, which includes many of the side tours discussed above, what remains is a reasonably simple-to-understand concept. There has to be a simpler way of discussing task activation than the daunting, intimidating version in the reference manual.

One more example of intimidating wording used for a relatively straightforward concept is 3.9:9. "If a subprogram declaration, a package declaration, a task declaration, or a generic declaration is a declarative item of a given declarative part, then the body (if there is one) of the program unit declared by the declarative item must itself be a declarative item of this declarative part (and must come later)." Again, if you wade through all this, you get to a simple concept. Which in this case is that program units consist of a declaration and a body that have to be declared in the same declarative region. There has to be a better way of saying 3.9:9 with acceptable rigor that is less obscure than what is in the current manual.

All I've done so far is only touch the surface of the problems you have with the Reference Manual. I've meant to do it with humor, even though the LRM's lack of clarity disappoints me greatly.

I wonder often, though, if sufficient attention has been paid to the human factors of this manual. I don't recommend that people actually use the LRM in their day-to-day code development; it's obviously not suitable to be used in that manner. As a language definition its lack of clarity is disturbing. Even though I normally am able to track down answers to questions, the process typically takes me far more time than it should. If I were to describe only briefly each of the problem areas of which I am aware, you would be presented dozens of pages.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Once people get up to speed on part of the language, that is what they use and they don't refer to the LRM.

POSSIBLE SOLUTIONS:

You need to hire a good technical writer and a good technical editor and a person with a good grasp of the workings of the language and have them spend many months hammering the LRM into a coherent and consistent and clear document. I'm not arguing for any changes in semantics in this request. But I think there's a real need for a more clear statement of what we have. The current document is in too many places an impediment to both the use of Ada and to the development of tools for Ada.

ORDER OF DECLARATIONS

DATE: September 29, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 IEN, UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 3.9, 9.1

PROBLEM:

The language imposes a partial ordering on declarations. The original intent was to encourage good style by ensuring that small items are declared first and do not get "lost" between large bodies which then follow. However, a subsequent change to the language allows a use clause as a later declarative item (this is very necessary) so that the original goal is not achieved. The current partial ordering is thus now restrictive for no good reason. For example, having declared a body, one cannot declare a representation clause.

Another curious restriction is that in a task specification, all representation clauses must follow all entry declarations. A similar rule does not apply to other declarative parts and prevents the natural style of pairing a representation clause with its entry declaration if there are more than one. This restriction may have arisen as an unwanted consequence of the syntax.

IMPORTANCE: ADMINISTRATIVE

A simple tidying up which would overcome the impression the language gives of knowing better than the programmer (that is being paternalistic) when instead it is merely an unnecessary inconvenience. The fact that this is not classed as "important" is no excuse for not giving it the attention it deserves. This would actually simplify the syntax and might make compilers a teeny-weeny bit faster.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

One possibility would simply be to eliminate the syntactic distinction between basic declarative item and later declarative item and just call everything a declarative item. And then give the rules of what is allowed by semantics. Note that there are lots of semantic rules in this area anyway (eg a deferred constant declaration can only appear in a package specification) and so this would not be a weakening of the description but a unification of its style.

If one wants to retain the distinction between those things that are not allowed in specifications (that is bodies) and those generally allowed everywhere, we could retain the class basic declaration (but having added use clause and representation clause to it) and then define

declaration ::=basic_declaration | **body**

declarative_part ::= {declaration}

Actually the whole syntax needs a good overhaul; it is riddled with partial semantics as all compiler writers know since they have to change it before it can be used with a parser; it might be worth asking all compiler developers for their views on how the syntax might be better expressed in order to see whether it could be revised into a more helpful form for them without losing pedagogic information for the normal user.

RENAMINGS AS BODIES

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 3.9

PROBLEM:

It is a reasonable and customary practice to write a subprogram specification before designing the body. The specification is not intended to indicate the manner of implementation, but only to indicate the interface, and, by comments, the effect. When a subprogram is provided by renaming an existing subprogram, this separation is impossible, since renaming is not a body.

The user of the subprogram should not care that it is provided by renaming, and thus should not see this fact in the specification. In writing a package, one wants to write in a specification:

procedure Increment(X: temperatures);

and then decide later in the implementation of the body that Increment should be a new name for an existing routine:

procedure Increment(X: temperatures) **renames** increase;

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Rename the subprogram in the specification or write a body which only calls the original subprogram. The first alternative violates information protection. The second is only efficient if the procedure call is expanded in line, but this is usually only supported when the bodies of subprogram being renamed and the one being exported are in the same compilation unit.

POSSIBLE SOLUTIONS:

Permit a subprogram renaming to be a subprogram body.

DECLARATIONS FOLLOWING BODIES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 3.9(2)

PROBLEM:

The language currently defines two classes of declarations that can appear in declarative regions: basic declarative items and latter declarative items. All basic declarative items must appear before any later declarative items in the region. For small declarative regions this is not very important, but for large declarative regions (such as the implementation of a package body) it can be annoying because types, constraints, variables, exceptions, etc that properly belong close to some particular latter declarative item must be separated from it due to any reason for it other than unwarranted concerns about the difficulty of compiler implementation.

IMPORTANCE: ESSENTIAL

This revision request is motivated primarily by concerns about symmetry and aesthetics, but it also yields some practical benefit to programmers because logically related entities within a given declarative region can now be kept together physically; this in turn makes programs more understandable and it makes it easier to find referenced entities (for example, variables global to a particular procedure inside a package body).

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Eliminate the distinction between basic declarative items and latter declarative items.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will re-compile successfully and will behave identically during execution except for possible small changes in execution speed.

PROGRAM ERROR RAISED FOR SUBPROGRAM ELABORATION

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 3.9 #5..8

PROBLEM:

The LRM requires checks to see if subprogram, generic, or tasks are elaborated. This is fine for dynamic first class objects, but it should not be required for a static case where items can be "pre-elaborated". Positive control is provided, but a good method for notifying the compiler system for pre-elaboration does not exist. Then, the compilation system should know when to elaborate at compile-time. Some compilers produce a map to say the elaboration order selected (if it can tell) and continues to process the subprogram. If the compilation cannot tell because the referenced subprogram is ambiguous or undefined, then the user should expect an error at the earliest point of recognition and not merely have the exception Program_error raised.

IMPORTANCE: IMPORTANT

This is especially needed for large scale developments with maximum reuse. For mostly static items, the compilation is known a priori and has very few dynamic cases or structures. The development team can not wait until the embedded computer is loaded and powered up to find a "program error".

CURRENT WORKAROUNDS:

Provide positive control of the elaboration of everything with a pragma. Execute the code under both the host, e.g., VAX, and the cross-compiled targets to assure strange exceptions are not being propagated for code-design errors.

POSSIBLE SOLUTIONS:

Provide the compiler with the capability to perform the static elaboration and have it produce a map for any elaboration decisions that it takes. Provide user warnings and information messages rather than waiting until execution time to raise Program_error.

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 6. SUBPROGRAMS

OVERLOADING "="

DATE: January 14, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 6.7(4)

PROBLEM:

It is not possible to overload "=" with parameters of different types.

CONSEQUENCES:

This prevents the overloading of "=" between the predefined type STRING and the user-defined type VARIABLE_LENGTH_STRING, preventing the implementation of a variable-length-string package which provides all the usual operators. Only the "=" between VARIABLE_LENGTH_STRING and STRING cannot be provided; all other operators between the two types can presently be provided.

More generally, any two types which one would reasonably expect to be compatible (bounded versus unbounded containers, for example) cannot presently be made compatible with respect to the "=" operator.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Delete the sentence "The explicit declaration of a function that overloads the equality operator "=", other than by a renaming declaration, is only allowed if both parameters are of the same limited type." from ANSI/MIL-STD-1815A 6.7 (4).

THE INABILITY TO HAVE PROCEDURES AS RUNTIME PARAMETERS OF PROCEDURES CAUSES PROBLEMS

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 6

PROBLEM:

There are two circumstances in which the inability to have procedures as run-time (as opposed to generic instantiation) parameters of procedures causes problems. Firstly, when interfacing to standard software packages such as X windows which make use of this style of programming. Secondly, when Ada is used as a target language by translators of higher level languages such as OPS5 and languages that support inheritance with run time type checking.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Compiler-specific involving manipulation of addresses and interfacing layers written in other languages (with resultant loss of type checking and exception handling)

POSSIBLE SOLUTIONS:

LRM DOES A POOR JOB OF DIFFERENTIATING SPECIFICATIONS AND DECLARATIONS

DATE: June 12, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 6, 7, 9, 12

PROBLEM:

This is an additional example of how the Reference Manual does an inadequate job of defining its basic terms. The concepts of declaration and specification are crucial to an understanding of the language. After working with the language for some time, one understands the different roles these two constructs play. However, we could make the lives of future students and compiler implementors easier if we were to provide in the reference manual a better explanation of these terms.

One shortcoming that I will not discuss further is that the glossary contains an entry for declaration, but not one for specification. This is odd, but not crippling.

The structure that Ada uses for the construction of programs from program units and to support separate compilation is relatively straightforward, but the amount of effort to learn that from the descriptions in the Reference Manual is too high.

Each program unit consists of two parts, its declaration and its body. The effect of compiling a declaration of a program unit is to make the external interface of that program unit available for use in constructing additional elements of the program library. The effect of compiling the body of a program unit is to place the implementation of that program unit into the program library for use in constructing executable programs. In less than six lines we've covered the essence of bodies and declarations. We realize that there are complications arising from the fact that tasks may not be separately compiled and such, but still we have to wonder why there is not a similarly coherent description of the model in the Reference Manual.

The best discussion is provided by the section on subprograms. LRM 6:2 "The definition of a subprogram can be given in two parts; a subprogram declaration defining its calling conventions, and a subprogram body defining its execution." Compare that to the definition of packages, LRM 7.1:1 "A package is generally provided in two parts: a package specification and a package body." And to tasks, LRM 9:4 "The properties of each task are defined by a corresponding task unit which consists of a task specification and a task body." Finally in LRM 12:2 "A generic unit is declared by a generic declaration." So what we have are two program units defined in terms of declarations and two program units defined in terms of specifications. Consistency in this area would be most useful.

Are these terms interchangeable, though? Clearly not. Subprogram declarations, package declarations and generic declarations are compilable. A specification is not. In every case the specification is not the construct that goes into the library. The specification is used to construct the declaration. For every

program unit except the package, the specification is repeated at the beginning of the body, presumably to allow for consistency checking.

IMPORTANCE:

If this were the only example of bad construction in the language, we could live with it. I, and many others, have pointed out additional examples of the Reference Manual's lack of readability or consistency. Taken as a whole, the Reference Manual is a poorly-written document. We should take the opportunity that the 9X work provides to improve the Reference Manual as a language definition so that those who implement Ada in the future or study the language in the future will have an easier time of it than we have had.

CURRENT WORKAROUNDS: NONE

Working with this language is more frustrating and costly than it needs to be.

POSSIBLE SOLUTIONS:

As part of the 9X work, engage a good technical writer and some Ada experts and charge them with clarifying the manual.

SEPARATE SPECIFICATIONS AND BODIES

DATE: July 10, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 6, 7, 9, and 10

PROBLEM:

The rationale for separating the spec from the body was to provide a definition of a unit that "management" can read. No "management" ever does. Also, the spec's have limited information--mainly just entry points and most information is duplicated in the body. Embedded systems have to prepare DoD Std 2167 documentation and the contents of an Ada spec has no useful information in it that would even aid the generation of one of the product specifications, e.g., SRS, STLDD, SDDD, SPS. Further, the recompilation rules are so expensive, that programmers are beginning to provide the WITHs on the bodies and not on the specs so that the specs do not have to be recompiled.

In addition, the program units are too fragmented. In most texts, it is impossible to write even a simple example of a program unit and get it on one page. This fragmentation cause the programmer to layer/nest procedures more than necessary. The "is separate" is buried far down in the code so that all the dependencies are hard to determine by the maintainer. Also, a programmer must keep track of at least 4 units and probably more, e.g, at least one package spec and body for the WITH and procedure spec and body. Most programmers keep them close together for visibility.

For another problem with the separation, the two compilation items have to be identified in the program library. One is just a fragment that is only useful to a compiler--the spec.

The next difficulty is order of elaboration problems with a spec inserted between a spec and a body that has additional initialization in the following body. There is an ambiguity on what to do. Most implementors elaborate the spec and then the body each time by creating a call to a runtime that links in the appropriate code. Some implementors treat elaboration as an INLINE and start copying code--very inefficient for embedded systems.

The additional lines of code (approximately 10% over languages that do not take this approach) for the structuring, separation, fragmentation, and duplication for separate spec's and bodies adds development costs over other comparable block structured languages--see Barry Boehm's Software Engineering Economics.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Not much encouragement for very complex programs. Much code has to be written just to give the compiler information for processing but does not generate useful executable code.

POSSIBLE SOLUTIONS:

It is only some academic interest for "pure" specs and bodies. There is no reason why these can't be merged.

IMPROVED INTERRUPT HANDLING

DATE: September 13,1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 6, 9, 13.5.1

PROBLEM:

The current Ada mechanism for handling interrupt is ambiguous, awkward and inefficient. It is conducive to undetected loss of interrupts and to timing errors

IMPORTANCE: IMPORTANT

A number of requests have been submitted by diverse organizations concerning problems with tasks. This request is intended to present a possible solution to some of those problems.

CURRENT WORKAROUNDS:

Modify the vendor's run-time support and handle interrupts in assembler procedures.

POSSIBLE SOLUTIONS:

Provide syntax for declaring a procedure to be a handler for a particular interrupt. Impose strict limits on what such a procedure is allowed to do, e.g., prohibit I/O. Provide a new type, "queue of record_type", a statement to post a record to the queue and a statement to associate the queue with a task entry.

An interrupt procedure should, in general, collect those data that would otherwise be lost, format them into a record, post the record to a queue and return with no further processing. Lengthier processing should normally be deferred to the associated task.

This proposal raises several questions

- Should the association of a queue with a task entry be static or dynamic?

- Should queues be available for more general use?
- Should the declaration of a queue include a size?

SUBPROGRAMS AS PARAMETERS AND FUNCTIONAL VALUES

DATE: July 24, 1989

NAME: D J Tombs (endorsed by Ada-UK)

ADDRESS: RSRE
St. Andrew Road, Great Malvern
Worcestershire WR14 3PS, UK

TELEPHONE: +44 684 895311

ANSI/MIL-STD-1815A REFERENCE: 6

PROBLEM:

A large number of programming problems imply the use of subprograms to other subprograms. The paradigm for this behavior is a numerical integration routine which takes as a parameter the function to be integrated. The integration algorithm will evaluate the function at several places within the domain of integration and deliver a weighted summation of the results. Another example is where the subprogram parameter implements a symbol table look-up.

Ada is inadequate for performing genuine data abstraction and for functional or object-oriented programming. To use such programming techniques a subprogram would have to be a first-class value which is manipulated just as a scalar and access value is: it would have type, and could be used to compose more complex values. This is not just an academic requirement of no practical benefit. Programming using these techniques was possible in ALGOL68 and is now coming into common usage with languages like ML and C++. As an example, at RSRE we are developing a simulator testbed for extensive use of homomorphic transformation techniques, and thereby seems impossible to write in Ada because the recursive function types which underlie the data structure of a transformation cannot be described.

Typically, nonrecursive algorithms which frequently need an expression to be evaluated have need of subprogram parameters. Function values occur in recursive algorithms
or where information hiding is important.

IMPORTANCE: IMPORTANT

To have subprograms as parameters borders on ESSENTIAL. Programmers are used to working in this manner in other languages, such as Pascal or C, and are surprised that a modern language like Ada cannot support this facility without a great deal of work.

It is not essential that subprogram values be implemented, since the average applications programmer may never encounter a need for them. They can still be regarded as important, since they are likely to be used more in the future.

CURRENT WORKAROUNDS:

It is possible to avoid subprogram parameters by using the generic mechanism. this is clumsy because generic subprograms must be instantiated before being used. Whilst this is a reasonable restriction for generics which introduce type polymorphism it is unnecessary for those which have only value and

subprogram parameters. Furthermore generics can yield inefficient code: most compilers implement generic instantiation by text substitution. Thus code is not shared between two instantiations, as it could be where the generic unit is not polymorphic.

There are no real workarounds for abstract data types and functional programming. Various proposals have been made linking private types, generic packages and tasks. Those that we value in the language (that is, one which can be treated as a data object), but it does not sit conveniently with the generic packages for polymorphism, information hiding and data abstraction.

POSSIBLE SOLUTIONS:

Is certainly possible to have another kind of formal parameter for subprograms, namely a subprogram specification. Such a parameter could only have mode in and a default value may be allowed; there are no procedure variables in Ada. As with other subprograms in Ada, those with formal subprogram parameters can be implemented on a stack.

At the subprogram call an actual subprogram must be supplies for the formal parameter. For consistency the matching rules can follow those given for renamed subprograms ((8.5(7.9) and for generic instantiation (12.3.6), namely that formal and actual must have the same parameter and result profile, and the parameter modes must match. Default subprograms should be allowed, with the matching rule that given for a generic formal subprogram in 12.36(3). Within the body of the main subprogram, if any call of the formal procedure, not the actual. Conversely, any constraint checks on a parameter value must be made against the subtype of the actual procedure, not the formal. Although these rules seem awkward, they are precisely those required to actual procedure parameter can be passed by "code reference"), whilst still being able to give the formal subprogram a helpful specification. A subprogram renaming declaration imposes precisely these rules and for the same reasons.

Thus subprograms as parameters can be implemented as a natural extension to Ada. As we have described them here they have a construction with same flavour as subprograms in Ada83, a well-defined (if sometimes obscure) semantic behaviors, and they can be implemented without necessarily implying extra heaped storage.

The extra syntax and semantics needed to incorporate subprograms as true values might be capable of definition without too many difficulties. There would have to be a means of declaring subprogram types, perhaps based on parameter and result profile, but otherwise the language is little changed. Likewise, matching rules could follow those described above. However a subprogram value must be represented in heaped storage: it must be called in a separate work space, and thus has similar allocation, assignment and deallocation problems as access values. Also, the subprogram call is less efficient. These complications are only necessary if the subprogram is anywhere treated as a value, eg assigned to a variable, and can, with difficulty, be optimized on other occasions. However, because of such problems, subprogram values may be considered too great a step to incorporate into Ada9X.

NEED WAY TO DECLARE SUBPROGRAM/TYPE/GENERIC IS SIDE-EFFECT FREE

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 6

PROBLEM:

There is no way to specify in Ada that a subprogram has no side effects, and have this specification checked at compile time, and used to advantage for optimization.

Any proof of correctness is made easier by side-effect-free subprograms. However, in Ada there is no way to securely define a side-effect-free subprogram. The same arguments apply to generic instantiations and default type initialization.

There are two kinds of side effects: side-inputs and side-outputs. A side-input is a reference to a variable entity external to the subprogram (generic, type definition). A side-output is an update to a variable entity external to the subprogram (generic, type def). Side-inputs prevent a subprogram from having an identical result given the same "direct" inputs. Side-outputs mean that the result of a subprogram is not captured by its in-out/out parameters or function result.

Either kind of side-effect will generally prevent an optimizer from removing what would otherwise be a redundant call on a subprogram, or moving it outside a loop.

Similarly, the default initialization of an otherwise unused object cannot be removed if there is a chance it has side-outputs. If it is known that the default initialization for a type has no side-effects, then it would be possible to construct an initial default value at type declaration time, and then simply copy it to initialize later default-initialized objects.

Finally, it is useful to be able to assert that a generic unit has no side-effects as part of its instantiation, and necessary for determining whether an enclosing subprogram has side-effects.

IMPORTANCE: ESSENTIAL

I believe this capability is essential to building large reliable systems, and ever having a hope of proving useful properties about them. Furthermore, it is essential to gain any of the potential benefits of "functional programming" to be able to treat certain computations as "pure" and side-effect free, amenable to lazy or cached evaluation strategies.

Without this capability code which in fact uses pure subprograms will not be optimizable to the maximum extent, and the benefits of functional programming will never be felt in the Ada world.

Such a capability requires language support for it to be securely and portably implemented.

CURRENT WORKAROUNDS:

There is no way now to declare a subprogram is side-effect free and have it checked.

POSSIBLE SOLUTIONS:

One possible solution to this problem is to define a language-defined pragma "PURE" (or perhaps "STRICT") specifiable on a subprogram, type, or a generic. It might also be useful to be able to specify a pragma PURE for a declarative region (such as a package), to assert that all subprograms, types or generics declared within this region are PURE. The meaning of pragma PURE for a subprogram would be as follows:

The body of the subprogram may not access or update the value or any alterable attribute of any variable object external to the subprogram itself, nor may it call any subprogram, instantiate any generic, nor declare a default-initialized object of any type defined external to the subprogram which is not declared to be PURE.

Objects designated by values of an access type declared external to a subprogram are considered "variables." (In general P.all may be considered equivalent to array indexing where the access type declaration implicitly declares the array object.) Local subprograms (types/generics) of a PURE subprogram need not be PURE relative to the PURE subprogram's variables, but they may not themselves access or update variables declared external to the enclosing PURE subprogram.

Note that PURE procedures are allowed to have IN OUT and OUT parameters, but these are the only variables which may be affected by the procedure call (i.e. no side-effects, but direct declared effects are legitimate).

The compiler will reject the body of a subprogram declared
to be PURE if it violates the requirements given above. Similarly the compiler will reject a type definition declared to be PURE if it calls any im-PURE functions as part of its default initialization, or contains any default-initialized components of im-PURE types.

A PURE task type is one which has no side-effects during its activation or execution, other than to its own local variables and to OUT and IN OUT parameters of its entries.

A PURE subprogram has the property that if its IN and IN OUT
parameters (and OUT parameter attributes) have the same value upon call, then the results will be the same upon return, and a compiler may eliminate such a redundant call if it recognizes it, and correctly simulates its result.

A PURE generic is one which contains no access or update reference to variables external to the generic during its elaboration (generic formal parameters are not considered external, and hence may be IN OUT mode).

NEED WAY TO SPECIFY USER PRE/POST CONDITIONS ON SUBPROGRAMS

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 6

PROBLEM:

In order to fully optimize in the presence of separate compilation, and to aid in the development of probably secure systems, it is essential that there be some way to further constrain the values of input parameters, and further characterize the values of output parameters in a way visible to all callers of a subprogram.

For example, as it is now, there is no way to specify that an access-value parameter must be non-null. This means that in the absence of pragma Suppress, a compiler must emit a constraint check on the first dereference of the access-value parameter inside the subprogram. Similarly, there is no way to specify that an integer must be non-zero (so divide need not check for zero-divide).

Similarly, there is no way to declare that a function returns a non-null access value, meaning that on first dereference of the result, again, a constraint check must be inserted.

It should be possible to augment the specification of a subprogram with sufficient pre and post conditions so that an optimizing compiler can safely eliminate *all* constraint checks from carefully written code.

Pragma SUPPRESS is *not* the solution to this problem. The right solution to this problem should allow a disciplined programmer to request that the optimizer issue a warning *every* time it is unable to eliminate a constraint check. Given this listing, the programmer should be able to insert additional pre/post condition specifications, and perhaps introduce additional subtypes, and be able to alter the program to the point where all constraint checks are eliminable.

IMPORTANCE: ESSENTIAL

This capability is essential to allowing safe Ada programs
to be fully optimizable, and to aid in efforts to create probably correct programs.

CURRENT WORKAROUNDS:

There are no good workarounds for the examples given above. They represent true holes in the user's ability to express the parameter requirements and result characteristics in the specification of the subprogram.

For an IN access-value parameter, it is possible to change the type to be the designated type and require all callers to dereference the parameter prior to call, pushing the constraint check to the caller (where it should be, ideally). For an IN-OUT parameter, or for the other examples given above, there does not seem to be any workaround.

POSSIBLE SOLUTIONS:

A possible solution is to augment the syntax for subprograms (or add a pragma) to allow the specification of a boolean precondition and postcondition. The precondition would be allowed to reference the values of IN and IN-OUT parameters. The postcondition would be allowed to reference the values of IN IN-OUT, and OUT parameters, as well as the "IN" value of IN-OUT parameters (perhaps using a syntax like param'IN), and the result of a function, perhaps as "<function>'RETURN."

Here is a possible syntax:

```
procedure <name>(<formal param specs>)
        in : <boolean expression>,
        out : <boolean expression, including "<in-out-param>'IN">;
```

and

```
function <func>(<formal param specs>) return <type-mark>
        in : <boolean expression>,
        out : <boolean expression including "<func>'RETURN">;
```

Alternatively:

```
pragma Precondition(<func>, <boolean expression>);
pragma Postcondition(<func>, <boolean expression>);
```

CONSTRAINT_ERROR would be raised by the caller if the precondition evaluates to FALSE. At the return statement, after evaluating the result expression if any, CONSTRAINT_ERROR would be raised if the postcondition evaluates to FALSE.

Presuming that the concept of side-effect-free expressions is added to Ada9X, then it would be required that the pre and postconditions have no side-effects. In any case, such side-effects would be erroneous, and the compiler may assume that no side-effects will occur as a result of evaluating a pre or postcondition.

PASSING PROCEDURES AS PARAMETERS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 6

SUMMARY:

The issue of procedures and functions as arguments in Ada has been pretty much beaten to death. Many Ada books show how to accomplish this functionality using generic procedures and functions. According to implementors, a mechanism similar to that used in Pascal can be used in Ada to achieve equivalent speeds of processing. Therefore, Ada and Pascal appear to be equivalent in power on this issue. Nevertheless, a readability issue remains regarding these generic subprogram solutions in Ada--they are simple not as readable and understandable as the equivalent programs which pass the subprograms explicitly. Therefore, the time has come to put subprogram parameters into Ada--even if the implementation is to immediately convert into generic subprogram form.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Many examples exist of programs which take functions as arguments, from the "mapping" and "reducing" functions of Common Lisp to the "Simpson's Rule" numerical integration programs of Fortran. Most modern languages offer some support for this concept, and the technical issues have all been worked out.

While Ada offers a mechanism through "generic subprograms" for achieving the same semantics as passing subprograms as arguments, these mechanisms are not nearly as readable or understandable as their Pascal counterparts.

We propose that Ada allow for the passing of subprograms as arguments to functions and subprograms in the manner of Pascal.

CURRENT WORKAROUNDS:

The Workarounds are well-known, but are still unsatisfying even if the efficiency is improved to be on a par with that of Pascal procedural arguments.

NON-SUPPORT IMPACT:

Unreadable programs and continuing arguments with computer scientists.

POSSIBLE SOLUTIONS:

Follow Pascal's example.

DIFFICULTIES TO BE CONSIDERED:

Professors will have to find other topics for final exams in their Ada courses.

This proposal is a direct violation of Steelman requirement 5D.

RECURSION

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 6.1

PROBLEM:

A note should not add semantics or syntax where the language is not specifically defined. In embedded systems, all programs are reentrant, but recursion is not allowed as the stacks can easily overflow the allocated space.

IMPORTANCE: Moderate

Recursion is seldom found in embedded applications as there is too high a potential for getting a storage space overflow.

CURRENT WORKAROUNDS:

Programming standards.

POSSIBLE SOLUTIONS:

<<minor impact>> Delete note in subpara. #10 in section 6.1 and any associated semantics that allow recursion without explicitly defining processes as being recursive.

SUBPROGRAM TYPES AND OBJECTS

DATE: September 27, 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraves AG
8052 Querich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 6.1, 6.3

PROBLEM:

The Ada language knows roughly 3 kinds of program units-packages, subprograms and tasks. Program units can be used (roughly) for two different purposes: Either as structural units (static nature) or as executive units (dynamic nature). Packages clearly are structural units whereas tasks clearly are executive units. It seems therefore reasonable, that Ada offers generic packages but no generic tasks, and task objects but no package objects. Subprograms on the other hand are kind of in-between. They may be used (and in fact are used) as structural units or as executive units, depending on the context. However, the executive nature of subprograms is only partially supported by the language. Whereas generic subprograms are allowed, subprogram objects are not.

The language should allow the declaration of subprogram types and subprogram objects. The language should allow to pass subprograms as subprogram and entry parameters.

IMPORTANCE: ESSENTIAL/IMPORTANT

ESSENTIAL: There is an urgent need for subprograms to be passed as parameters (only tasks can be passed as parameters, at the time). Due to the asymmetric nature of Ada tasking, it is sometimes necessary to introduce two intermediate messenger tasks to work around the problem.

IMPORTANT: Subprogram types and subprogram objects are something quite common in other programming languages. Many now involving tasks and generic instances could be greatly simplified, thereby reducing compile time and/or run time overhead. Subprogram objects would also strengthen the object-oriented nature of Ada (c.f "dynamic binding").

Moreover, the Ada software developer is tempted today to use implementation-dependent or non-portable features.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Declaration of subprogram types, subprogram objects and single subprogrmas must be allowed. Two subprograms are assignment compatible, if they have the same parameter and result type profile. Explicit

conversion between subprograms having the same parameter and result type profile is allowed as well.

```
PROCEDURE TYPE p_type       (...);                      -- procedure type
FUNCTION TYPE f_type        (...) RETURN some_type;     -- function type

PROCEDURE p : p_type;       --this is a procedure constant declaration
FUNCTION f : f_type IS      --this is a function constant body

BEGIN
  --
END;

p_p : p_type :=P;                   -- variable procedure object
f_f : CONSTANT f_type:=f;-- constant function object

PROCEDURE single_p (...);-- single procedure declaration
                         -- of an anonymous procedures type

FUNCTION single_f (...) RETURN some-type IS
                                    -- single function body of
                                    -- an anonymous function type

END;
```

The suggested extensions are strictly upward compatible.

VARIABLES OF "SUBPROGRAM" TYPE SHOULD BE SUPPORTED

DATE: October 19, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 6.1

PROBLEM:

There are many situations where it is efficient to store the "address" of a subprogram in a table and at some later time call the subprogram.

There are also circumstances where it would be useful to pass the "address" of a subprogram to a non-generic subprogram. This is particularly important when interfacing to separate subsystems (such as a database manager or a windowing system), where it is not practical to implement the interface to the subsystem using generics since a Pragma Interface is being used.

Storing the address of a subprogram, and passing it to another subprogram should be possible without resorting to unchecked programming or implementation-dependent features of the language, while preserving strong type/interface checking.

IMPORTANCE: ESSENTIAL

If this capability is not provided, programmers will continue to be forced to use non-portable, implementation-dependent, and type-unsafe mechanisms for storing and passing subprogram addresses.

CURRENT WORKAROUNDS:

The 'ADDRESS of a subprogram may be taken, and stored or passed to another subprogram. The subprogram may be called either by resorting to assembly language, or by utilizing an implementation-dependent interpretation of the subprogram address clause to associate the address with a locally declared, pragma-interfaced subprogram.

POSSIBLE SOLUTIONS:

Storing the address of a subprogram presents a slightly different problem from passing the address to another subprogram. When storing the address, it is essential that the address not later be called when the subprograms enclosing scope (if it is nested) is no longer available.

When passing the address of a subprogram, this is not a problem, and ideally even a reference to a nested

subprogram should be passable to a subprogram in which it is not statically nested. In this case, it is essential that a reference to its enclosing scope be provided in addition to the code address of the subprogram (i.e., a static link, or equivalent).

Because these two represent different problems, I will propose slightly different solutions:

A. Non-Limited Subprogram Types

When storing the address of a subprogram, it is essential that the stored address not "outlive" the subprogram. This is analogous to the access type problem, which Ada solved by introducing an explicit intermediate type, the "access type" rather than having anonymous pointer types.

Here is a possible solution:

<type_declaration> ::=
procedure type <identifier> [formal_part];

| **function type** <identifier> [<formal_part>] **return** <type_mark>;

These declarations declare non-limited types, for which assignment and equality are defined. Instances of these types may be declared, assigned, called, passed as parameters, and converted.

Conversion is defined such that an instance of a subprogram type may be converted to another subprogram type if the parameter profiles match (see below) and if the target type is declared somewhere within the scope of the source type, or within the same declarative region as the source type. The scope rule is intended to ensure that upon conversion to the target type, the subprogram "value" can never be stored in a variable accessible at a point when the enclosing scope of the subprogram is not available.

The parameter profiles are required to match according to the following rule:

> The two subprogram types must have the same number of parameters of the same base types and same parameter modes. The parameter subtypes must identify the same subtype unless both are static or unconstrained subtypes. If both are static subtypes, then the constraints must match. The return subtypes must match in the same way.

This rule ensures that the caller may perform constraint checking based on the parameter profile of the target type and know that no constraint violations are being missed.

Instances of a subprogram type are initialized to a null value by default, and a call on a null subprogram variable raises PROGRAM_ERROR.

B. Limited Subprogram Types

It is also useful to support a "limited" subprogram type. Instances of this type may not be assigned or compared for equality. However, no scope limitations exist for conversion. An instance of one subprogram type (limited or non-limited) may be converted to any limited target subprogram type so long as the parameter profiles match (as defined above).

Limited subprogram types are ideal for use as formal parameters, since they allow any instance of a matching subprogram type to be passed as an actual. They are not useful for declared objects, since they cannot be initialized nor assigned.

A possible syntax for limited subprogram types is:

<type_declaration> ::=
limited procedure type <identifier> [formal_part];

| **limited function type** <identifier> [<formal_part>] **return** <type_mark>;

C. Subprogram "literals"

"Normal" subprogram declarations are treated like declarations of overloadable literals of an anonymous "universal" non-limited subprogram type, which are implicitly convertible to any matching limited subprogram type, or to any matching non-limited subprogram type with compatible scope (as defined above). As with other literals, conversion is only a last resort if the expression is meaningful without the conversion.

Note that an instance of a non-limited subprogram type may be initialized from a previously declared subprogram (literal) so long as it matches and is legally convertible to the specified subprogram type.

D. Parameterless (or fully defaulted) function types. There is a syntactic ambiguity with parameterless (or fully defaulted) function types, since it is not clear when objects of such a type are referenced whether the function is to be called or the object itself is being referenced.

The simplest solution is to use context to resolve the problem, with the default being that the function is called in "no context" situations (i.e. a conversion). This is backward compatible with Ada '83 because parameterless functions are always called when they are the source of a conversion.

SUBPROGRAMS AS PARAMETERS OF SUBPROGRAMS

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 6.1

PROBLEM:

In many applications specific problems can only be supplied by procedures or functions where a problem-solving module uses calls of these user-supplied subprograms to update iteratively the contents of problem-specific workspace. A simple example is a general zero-finding subprogram for which the (continuous) function f : R --> R is provided by an Ada function f(x) that delivers f-values for feasible values of the argument.

In most cases the recourse to a generic subprogram is not possible because of its restriction of staticness, whereas in many user-supplied subprograms the execution will depend on information that is only available dynamically.

The general scheme is (using the generics concept):

```
generic
   with function F(COORDINATES : COORD_TYPE) return FLOAT;
procedure SOLVER (parameters);
                  -- for bounds, tolerance, step size, etc.
```

Examples:

a. In one of our large-scale Ada implementations the solver was a method for integrating elliptic partial differential equations by a multi-grid method where switches took place between several levels of coarse and finer grids. The iterations on every level (implemented by subprograms) could be standard (locally provided) or user-defined, for which the obvious approach was a generic parameter. Therefore, the user has to make the generic instance (this gave the problem that the locally provided standard iteration had to be library-provided).

   The transitions between two successive levels were all similar and therefore (for every pair of levels) also obtained by generic instantiation with suitable parameters. This instantiation could not be combined with the one previously described, thus resulting in another level of instantiating. The instantiation should use one of the instances mentioned above, or alternatively, one parameter of the latter generic module should be allowed to be a GENERIC subprogram itself (which is not allowed).

Finally, the problem definition F (or F1, F2, ..., FN in the case of several problems) required a third level of instantiating, with the drawbacks of generic instantiating to be mentioned below. With subprograms as parameters of subprograms these drawbacks should have been avoided.

b. Another example where the need to use generics for the passing of subprograms proved to be very troublesome is:

One of our packages contains procedures to carry out basic arithmetic operations on combinations of vectors, scalars and matrices. The four basic operations applied are multiplication, subtraction, addition, and division. These are applied to 10 classes of object combinations:

real scalar	-- real vector
real scalar	-- real matrix
real vector	-- real vector
real matrix	-- real matrix
real scalar	-- complex vector
real scalar	-- complex matrix
real vector	-- complex vector
complex vector	-- complex vector
real matrix	-- real matrix
complex matrix	-- complex matrix

This represents 10 classes of subprograms with 4 members per class for a total of 40 subprograms. Within each class the code is very similar. All that varies is the operation applied to pairs of elements. We implemented the 10 procedure classes within a generic package. This package contains a generic formal parameter for the operation to be applied between elements. We then instantiate the package for each arithmetic operations to yield the desired subprograms. This works well but it has some drawbacks.

1. Generic Instantiation is slow and is done at run time.

2. Code size is huge - especially with compilers that do not utilize code sharing for generics.

3. It is somewhat awkward to implement (an admittedly subjective view).

In our opinion, a much cleaner approach would be to write 10 subprograms, one for each class, and have as a parameter to the subprogram a subprogram representing the arithmetic operation of interest. This would alleviate each of the above drawbacks.

IMPORTANCE: IMPORTANT

This requirement is ESSENTIAL to everyone, not only to all implementors of Numerically Intensive Computing (NIC) methods. In other words, the revised standard is unlikely to be accepted by the community making the suggestion if this revision request is not satisfied.

CURRENT WORKAROUNDS:

Offered solutions, such as reverse communication and the use of generics, only work for the simple cases, they lead to opaque source code and are likely to introduce safety gaps.

POSSIBLE SOLUTIONS:

1. Subprogram parameter specifications (but not necessarily also those of task entries) should allow a formal procedure/function designator together with a parameter and result type profile, which can be associated with a declared subprogram that has a corresponding parameter and result type profile.

2. Such a feature might imply the availability of a way of declaring a 'function type' and a 'procedure type', simply for the purpose of being able to write parameter specifications. That such subprogram types might then also be used for declaring individual subprograms (and, e.g., arrays of subprograms) was a possible additional feature that did not receive overwhelming support by our community.

   A possible syntax for subprogram types might be:

```
subprogram_type_declaration ::=
        procedure type identifier [formal_part] ; |
                function type designator [formal_part] ; |
                        return type_mark ;
```

   Examples:

```
function type F_TYPE (A : in A_TYPE; B : in B_TYPE)
return R_TYPE;

procedure type P_TYPE (A : in  A_TYPE ;
                       B : in  B_TYPE ;
                       C : out C_TYPE );
```

Then a procedure declaration with a subprogram parameter could be:

```
procedure PASS_P ( P : P_TYPE );
```

PROCEDURE AND FUNCTION TYPES

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 6.1

PROBLEM:

Ada does not provide a procedure type or a function type. Therefore, procedure and function references cannot be passed in subprogram calls.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Allow procedure and function types and objects similar to task types. Permit the assignment of these objects and the execution of them. Require the same matching rules as for the subprogram renames. Do not permit an object to be overloaded (that would make the assignment too complex).

DEFENSIVE PROGRAMMING

DATE: August 27, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 6.1.2

PROBLEM:

When engaging in defensive programming, it is frequently the case that subprogram parameters are subjected to specific tests which, if not satisfied, result in the raising of some sort of exception.

It is also frequently the case that the nature of the testing is such that the testing could generally be done at compile time for statically specified actual parameters.

Presently, there is no means by which one can specify validation procedures which are attached to specific formal parameters, and a compiler is therefore forced to generate code which will always perform the validation at run time, rather than engaging in the compile-time verification of statically specified actual parameters and subsequently generating code which skips the validation sections and proceeds directly into the functional processing.

IMPORTANCE:

CONSEQUENCES:

The practice of defensive programming, which is to be strongly encouraged, is made costly due to inherent limitations on what can be done at compile time under the definition of Ada 83.

Also, the detection of parameter validation errors is deferred to run time, rather than compile time (where error correction is much less costly), which drives up the cost of Ada software development.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Design a mechanism which will permit the attachment of a specific validation procedure to each formal subprogram parameter, such that compilers are capable of performing the validation at compile time rather

than at run time in the (frequently occurring) situations in which the actual parameter can be statically determined. There should also be the capability of enforcing inter-parameter constraints (i.e., constraints which govern the relationship between two or more actual parameters); these constraints could be checked at compile time if all of the actual parameters involved could be determined in advance.

CANNOT DEFINE AN ASSIGNMENT OPERATOR FOR A LIMITED PRIVATE TYPE

DATE: August 8, 1989

NAME: J R Hunt

ADDRESS: Flessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 6.13

PROBLEM:

Cannot define an assignment operator for a limited private type.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Define appropriate copy procedures.

POSSIBLE SOLUTIONS:

Allow a procedure ":=" to be declared.

PARAMETER PASSING MECHANISMS

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 6.2

PROBLEM:

There is an unfortunate interaction between generics and the prescription of the parameter passing mechanism. Consider

```
generic
        type T is private;
procedure foo(X: in out T);
```

Depending on the instantiation, X will have to be passed by value or can be passed by reference. As a consequence, code sharing for this generic procedure becomes more difficult: two implementations need to be generated (assuming that passing large matrices by copy will not be acceptable to the Ada user).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

It should be considered whether the distinction between scalar and other parameters can be dropped in the determination of the parameter passing mechanism.

Should, in reaction to other comments, the language be refined to allow the specification of the specific parameter passing mode on a per-parameter basis, then it should be seriously considered to allow implementations complete freedom of choice in the absence of such a user-provided prescription.

PERMIT READING OF OUT PARAMETERS

DATE: August 27, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne, IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 6.2(5)

PROBLEM:

A subprogram which returns a value as an out parameter cannot read any value it gave the out parameter and is therefore forced to keep a second copy. This wastes time and space.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use a local variable for the out parameter, assigning the value of the local variable to the out parameter before exiting the subprogram.

POSSIBLE SOLUTIONS:

Allow the reading of an out parameter by the subprogram which is generating its value. It should be possible at compile time for the compiler to detect and report any reads which occur before an update is made to the out parameter.

UPDATES TO FORMAL PARAMETERS OF OUT MODE

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 6.2(5)

PROBLEM:

Updates to formal parameters of mode out

Is the rule that forbids reading out parameters strictly necessary, or is it a way to protect against reading undefined values?

If it is only for protection, then it is unclear why this protection is not general. A simple general protection mechanism against undefined values (at the cost of double (virtual) memory) could store for each object its value and unequally large bistring that indicates defined or undefined value.

If this kind of general protection mechanism would be supported (which is up to the compiler vendors and not to the language designers), then one might ask why reading of out parameters would not be permitted, after all, reading of local variables is exactly as "dangerous" as reading out parameters.

The basic problem with rule 06.02(05) is not only that it is not a generally applicable protection against undefined values. Anywhere one would read an out parameter if rule 06.02(05) did not exist, one needs to assign the out parameter to a local first, and subsequently reading the local. But this approach entirely removes the protection since forgetting the assignment results in reading an undefined (and typically history dependent) value.

The problem can be demonstrated by an example, which attempts to illustrate that principles like "do not use out parameters as local variables" sound very noble, but fail in practice to reduce the likelihood of error or human misinterpretation. This is not meant as an encouragement to use an object as a temporary variable, just to avoid introducing a new declaration (for saving bytes, god forbid). Such very old fashioned "tricks" violate software engineering principles, but there is nothing the language can do to avoid this (can still be freely practiced using object declarations or in out parameters), so it shouldn't attempt to avoid it just for out parameters, unless there are other justifications than the sick practices mentioned above.

This attitude seems to be conform with the Ada rationale, which nicely quotes Orwell's "1984" to defend this principle, by stating that the language designer should never take the attitude of the Newspeak designer:

> "Don't you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thought-crime impossible, because there will be no words in which to express it."

Assuming that the "crime" mentioned above is not forbidden by the language (but by education for example), one should concentrate on all other use of the language to evaluate the language's semantics. So, re-use of an object (in the case of rule 06.02(05) an out mode formal parameter) must imply that this object represents something that is to be built stepwise. A simple example to illustrate this issue outside the scope of 06.02(05):

```
declare
        TOTAL:NATURAL
begin
        TOTAL := 0;
while SOME_CONDITION
loop
        TOTAL := TOTAL +1;
        SOME_OPERATION;
end loop;
        TEXT_IO.PUT_LINE (The total number of ... is" & NATURAL'IMAGE (TOTAL));
end;
```

I don't think many people would argue that this code is bad because the variable TOTAL was used to hold a value that should be called TOTAL_SO_FAR while being calculated, to be assigned to a variable TOTAL immediately after the loop. And probably even less people would argue that it is a language requirement to try to avoid such coding style. And those who do, should think about the impurity of overloading the name TOTAL_SO_FAR for the concepts TOTAL_INITIAL, TOTAL_AFTER_ONE, etc..., and might soon realize that there is always a practical limit to the simple "one concept, one name" principle. One should further realize that the more subtle declarations get, the more likely they will be confused with another, closely related, declaration.

So, let's now look at the example announce above that illustrates the weakness of rule 06.02 (05):

```
--Ada83 version
procedure TRY_TO_DO_IT (INPUTS : T;
                                    SUCCESS: out BOOLEAN;
                                    RESULT : out T) is
  PARTIAL_SUCCESS : BOOLEAN; -- Local to avoid trouble with 06.02(05)
  PARTIAL_RESULT :T;
Begin
        TRY_TO_DO_PARTIAL (INPUTS, PARTIAL_SUCCESS, PARTIAL_RESULT);
--If the previous line would have erroneously used SUCCESS instead of PARTIAL_SUCCESS
--the nest line will read an undefined value PARTIAL_SUCCESS without any compiler
--warning

if PARTIAL_SUCCESS
        then  TRY_TO_DO_PARTIAL_REST  (PARTIAL_RESULT,  PARTIAL_SUCCESS,
        RESULT);
end if;

SUCCESS := PARTIAL_SUCCESS;
--If one would forget this line, the caller will probably read an undefined value
end;
```

The former example (and certainly in larger examples of the same style) invites people to forget the final

assignment to the out parameters or to use the formal parameter where the local variable should be used (in update operations), both as harmful as reading undefined values since both errors make the program erroneous without compiler notice.

So, wouldn't it be preferable if Ada9X would allow:

```
--Ada9X proposal
procedure TRY_TO_DO_IT (INPUTS :T;
                        SUCCESS: out BOOLEAN;
                        RESULT: out T ) is
PARTIAL_RESULT : T;
begin
TRY_TO_DO_PARTIAL (INPUTS, SUCCESS, PARTIAL_RESULT);
if SUCCESS -- Illegal Ada83
        then TRY_TO_DO_PARTIAL_REST ( PARTIAL_RESULT, SUCCESS, RESULT);
end if;

end;
```

In the (rather typical) example above, one gets obviously no protection at all from the rule under consideration: for each possible programmer mistake in the latter example introduced by reading the formal out parameter before assigning to it, one can think of the equivalent error in the former version by reading the local variable before assigning to it.

IMPORTANCE: ADMINISTRATIVE

As a single instance of the problem. The general approach of trying to half-protect against undefined values is IMPORTANT.

CURRENT WORKAROUNDS:

Explained together with the problem.

POSSIBLE SOLUTIONS:

Let out parameters only indicate to the user and to the compiler that initial values of the actual don't matter, the subprogram is committed to assign a value to the argument before it is read, either by itself of by its caller (same commitment as the one implied for local variables).

High quality compilers that support runtime checks for undefined values will hopefully obviate a lot of language rules that try to avoid undefined value access.

Suggested modification does not imply any upward compatibility problems.

It might be noted that language restrictions should be justified with respect to the design goals, which contain "above all, an attempt to make the language as small as possible" and "concern for programming as a human activity", and not for instance "things should look symmetric" or "there should be enough difference between in out and out parameters" Symmetry or semantic distance are not goals of their own, and besides the proposal is not to ignore the different semantics between in out and out modes: for the callee all remains as is, for the callee it only asks for consistent treatment of the same problem: in out parameters should be considered initialized variables and out parameters should be considered uninitialized

variables, hence reading out parameters before assigning them is erroneous and not illegal.

READING OF OUT PARAMETERS THAT ARE OF ACCESS TYPES

DATE: October 22, 1989

NAME: Arnold Vance

ADDRESS: Afflatus Corp.
112 Hammond Rd.
Belmont, MA 02178

TELEPHONE: (617) 489-4773
E-mail: egg@montreux.ai.mit.edu

ANSI/MIL-STD-1815A REFERENCE: 6.2(6), 3.2.1(10)

PROBLEM:

Implied by 6.2(6), an out formal parameter that is of an access type will have the current value of the actual parameter at the time of the call to the procedure. If Revision Request 0002 is implemented, it would be erroneous to read this value before it was overwritten (i.e., initialized). However, it would be safer (and consistent with access type objects that are not explicitly initialized) to implicitly initialize it to the null value.

IMPORTANCE: IMPORTANT

Should help reduce the degree to which programs are erroneous.

CURRENT WORKAROUNDS:

Explicitly initialize the access type formal parameter before use. If forgotten or otherwise left out, the subprogram may have access to the designated object of the access value of the actual parameter.

POSSIBLE SOLUTIONS:

Require that out parameters of an access type be implicitly initialized to null. Even if 0002 isn't implemented this is probably a good change: from the point of view of the variable that is the actual parameter, it should be similar to initialization when the object was created (explicit initialization corresponds to the procedure updating the formal, implicit initialization corresponds to the procedure not updating the formal).

UNITIALIZED OUT MODE ACCESS PARAMETERS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 6.2(6)

PROBLEM:

For purposes of constraint check elimination, it would make things simpler if one could always assume that immediately after a successful call to a procedure in which an object was passed as OUT mode parameter, the object must satisfy the constraints of the subtype of the corresponding formal parameter.

Given the example,

```
type T is ...;

subtype S is T ...; -- some constraint

X : T;
Y : S;

procedure P (Z : out S is ...;
begin
...;
P (X);
Y := X;
```

one might think that no constraint check would be needed for the assignment of X to Y, regardless of how the elided regions of code in the example are to be filled in. Unfortunately, this would be incorrect. For example,

```
type T is access String;
subtype S is T (1.. 10);

X : T;
Y : S;

procedure P (Z : out S) is
begin
        null;
end;
begin
```

```
X := new String (1 ... 5);
P (X);
Y := X;
```

Because 6.2(6) requires copy-in for access parameters of mode OUT, and because the procedure P never stored another value into its parameter, the value of X does not satisfy the constraint of the subtype S immediately after the call to P, so the assignment to Y should raise the Constraint_Error.

To address this, 6.2(6) should be amended to require, or at least give implementations the option of, initializing OUT mode parameter of an access type to null instead of copying in the previous value of the parameter. It is certainly true that this introduces an inconsistency between the treatment of access class parameters and of access class subcomponents of array or record class parameters, but the parameter passing rules for records and arrays are already quire different from those for scalars and access values, so this does not deem like a problem.

The constraint check elimination problems introduced by the present treatment of access class OUT mode parameters become more serious in the context of shared code generics.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

VERIFICATION OF SUBPROGRAM PARAMETERS

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 6.3

PROBLEM:

The Reference Manual does not require that a subprogram parameter be used within the subprogram body.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

No workarounds in the language - this check is either done manually or with some software tool.

POSSIBLE SOLUTIONS:

Include a requirement in the Reference Manual to make this check.

ALLOW INSTANCES OF GENERIC SUBPROGRAMS AS SUBPROGRAMS BODIES

DATE: September 28, 1989

NAME: David F. Papay

ADDRESS: GTE Government Systems Corp.
PO Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 6.3

PROBLEM:

Currently the standards requires subprograms bodies to follow the syntax and semantics described in section 6.3. As others have no doubt requested, it would be desirable to allow renaming declarations to take the place of a subprogram body. I would like this idea expanded to allow instances of generic subprograms to be permitted at places where subprogram bodies would be required. If such a subprogram were declared within a package, users of the package would see only the specification. The instantiation would appear in the package body and would serve as the body for the corresponding specification. This would hide the use of generic users from the user. For example:

```
package EXAMPLE is

        type MY_TYPE is TBD;
        type MY_PTR_TYPE is access MY_TYPE;

procedure FREE (X : in out MY_PTR_TYPE);

end EXAMPLE;

with UNCHECKED_DEALLOCATION;
package body EXAMPLE is

procedure FREE is new UNCHECKED_DEALLOCATION
        (OBJECT => MY_TYPE,
        NAME   => MY_PTR_TYPE);

end EXAMPLE;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Declare instantiation in the package specification; Declare a simple "shell" procedure which in turn calls

the instance.

POSSIBLE SOLUTIONS:

As described above.

PROVIDE T'STORAGE_SIZE FOR TASK OBJECTS (SIGADA ALIWG LANGUAGE ISSUE 37)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

Currently, in order to specify "the number of storage units to be reserved for an activation (not the code) of a task" [LRM 13.2(10)], an Ada programmer must define a task type. This rule prevents a programmer from defining a unique task object and setting the storage size for the task.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

(1) Create a package which defines a procedure interface that matches the task interface and in the body of the package declare a task type; set the Storage_Size for the task type, declare a task of the task type using an object declaration and call the task entries from the procedures declared in the package. This technique results in unnecessary work on the part of the Ada programmer (introducing the potential for error) especially if there is a need for timed and conditional calls to the task entry.

POSSIBLE SOLUTIONS:

1. Add the capability to specify the Storage_Size of a task object declared using a task_declaration (only).

   Change LRM paragraph 13.2(10) to read:

   The prefix T must denote a task type [or a task object created by a task_declaration]. The expression must be of some integer type (but need not be static); its value specifies the number of storage units to be reserved for an activation (not the code) of [the specified task or] a task of the type.

2. Add the capability to specify the Storage_Size of a declared task object created by a task_declaration or an object declaration (not a parameter of a task type or a renaming declaration).

   Change LRM paragraph 13.2(10) to read:

The prefix T must denote a task type [or a task object created by a task_declaration]. The expression must be of some integer type (but need not be static); its value specifies the number of storage units to be reserved for an activation (not the code) of [the specified task or] a task of the type.

The semantics must be defined for the situation where a programmer attempts to specify both the Storage_Size of a task type and an object of that type.

This issue was discussed at the August 1987 ALIWG meeting where it was supported by a vote of 12-4-0 (for/against/abstain) in favor of specifying storage size for a task of anonymous type. A related question was also supported, which is the ability to specify different storage sizes for different objects of the same task type.

ADD REPRESENTATION ATTRIBUTES TO ENUMERATION TYPES (SIGADA ALIWG LANGUAGE ISSUE 36)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

While it is possible to define the representation of enumeration values, there is no implementation-independent mechanism to find the representation of a enumeration literal or to find the enumeration literal which corresponds to a representation value.

This issue was discussed at the December 1987 ALIWG meeting, where it was approved by a vote of 11 to 8.

IMPORTANCE: IMPORTANT

The lack of this capability is inconsistent with the remainder of the language, where attributes for types and objects are defined to provide the user with access to information that typically part of the internal representation during compilation or execution (e.g., 'First, 'Last, 'Image for discrete types, or 'Size for constrained types), or that can be determined in by the compiler or the run-time environment ('First, 'Last and 'Length for array objects, or 'Size for dynamically sized objects)

CURRENT WORKAROUNDS:

1. Use Unchecked_Conversion to convert from the enumerated type to some integer type (and the converse). However, this mechanism is not portable as a result of implementation differences in the representation of enumerations types and in the treatment of unchecked conversion, as described in the following paragraphs from the LRM.

    a. "In the absence of a representation clause for a given declaration, a default representation of this declaration is determined by the implementation." [LRM 13.1(5)]

    b. "An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware. [LRM 13.1(10)]

    c. "An implementation may place restrictions on unchecked conversion, for example restrictions depending on the respective sizes of objects of the source and target type." [LRM 13.10.2(2)]

This mechanism also requires the user to define one instantiation of Unchecked_Conversion for each conversion to be performed (e.g., Integer -> Enumerated Type, Enumerated Type -> Integer, Short_Integer -> Enumerated_Type, etc.).

2. Define arrays or functions which convert from the enumerated type to some integer type (and the converse).

e.g. 1

```
type Color_Type is (Red, Yellow, Blue);
for Color_Type use (Red => 1, Yellow => 2, Blue => 3);

Integer_Representation_of_Color: constant array (Color_Type) of Integer :=
            (Red => 1, Yellow => 2, Blue => 3);  Color_Value_of_Integer: constant array (Integer
range 1 .. 3) of
Color_Type :=
            (1 => Red, 2 => Yellow, 3 => Blue);
```

e.g. 2

```
type Color_Type is (Red, Yellow, Blue);
for Color_Type use (Red => 1, Yellow => 10, Blue => 100);
Integer_Representation_of_Color: constant array (Color_Type) of Integer :=
            (Red => 1, Yellow => 2, Blue => 3);

function Color_Value_of_Integer (Integer_Value: Integer) of Color_Type is

begin
      case Integer_Value is
            when 1 =>
                  return Red;
            when 2 =>
                  return Yellow;
            when 3 =>
                  return Blue;
            when others =>
                  raise Constraint_Error; -- or some other exception
      end case;

end Color_Value_of_Integer;
```

However, this approach is subject to user error if there is a change in representation and may also be non-portable depending on the types involved.

POSSIBLE SOLUTIONS:

Define the following new attributes for all enumerated types (or all discrete types to be consistent with the treatment of other attributes for enumerated and integer types - see LRM 3.5.5):

P'Representation For a prefix that denotes a discrete type:

This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type Universal_Integer. The result is the internal representation of the parameter X.

P'Literal For a prefix that denotes a discrete type:

This attribute is a function with a single parameter. The actual parameter X must be a value of the type Universal_Integer. The result type is the base type of P. The result is the literal value in the base type of P whose internal representation is the Universal_Integer value corresponding to X. The exception CONSTRAINT_ERROR is raised if X does not correspond to the internal representation of any of the literal values in the base type of P.

DEFINING FINALIZATION FOR OBJECTS OF A TYPE (SIGADA ALIWG LANGUAGE ISSUE 35)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

There is no way to define actions to be automatically performed on an object when leaving the scope of the object. This results in the incomplete definition of an abstraction. For example, there is no way to automatically the free dynamically allocated memory created using an access type (or a user defined memory-managed type) when leaving the scope an object; oı to release a locked resource associated with an object of a user defined type. The lack of this capability can cause memory fragmentation or loss of other resources during the execution of a program.

This issue was first discussed at the August 1987 ALIWG meeting, where it received overwhelming support.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

1. Define a procedure that a user of a type calls before exiting the scope of an object of the type (e.g. instantiate Unchecked_Deallocation). However, this approach is unreliable since it puts the burden of correct behavior on the user of a type, rather than the definer of the type.

2. For most predefined types, compilers automatically perform a predefined operation when exiting the scope of an object of the type (e.g., freeing the memory associated with the object, synchronizing with dependant tasks, etc.).

POSSIBLE SOLUTIONS:

1. Define for every type a predefined "Finalize" operation. The syntax of the operation for most types would be the following:

   procedure Finalize (Item: **in out** Item_Type);

   However, in the case of task types (an task objects of anonymous types), the syntax of the operations should be an entry of the form:

```
entry Finalize;
```

The Ada programmer should be able to hide the predefined "Finalize" operation with a user defined operations (just like the user can hide a predefined operator for the type) by declaring a procedure (or entry), instantiating a generic, or procedure renaming.

When exiting a the scope of an object the compiler should call the appropriate "Finalize" operation for the type of the object prior to performing any actions that it normally performs for an object of this type.

Note that much of the compiler logic required to support this already exists, because of the semantics of leaving a block in which tasks are declared. At the August 1989 meeting it was pointed out that the terminology "end of scope" could be sharpened somewhat, perhaps using the term "life of the object". For example in the following, the finalization operation should not be applied to object X until the procedure Write_To_Disk is finished with it:

```
function F returns T is
        X : T := whatever;
begin
        return X;
end F;
...
Write_To_Disk( F);
```

2. Define a pragma which a user can apply to a procedure or entry which informs the user that a specific operations is the "Finalize" operation.

   e.g.

```
type T is ....;
procedure Kill (Item: T);
pragma Finalize (Type_Name => T, Operation_Name => Kill);
```

Note: the chapters and sections listed under references would need to be modified to accommodate the addition of this operation.

TYPE RENAMING
(SIGADA ALIWG LANGUAGE ISSUE 33)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

Paragraph 8.5(16) states in the form of a note: "A subtype can be used to achieve the effects of renaming a type (including a task type) as in

```
subtype Mode is Text_IO.File_Mode;"
```

However, having visibility to the subtype does not guarantee visibility to the operations or values of the type.

Example

```
with Text_IO;
package Simple_IO
is
        subtype File_Type is Text_IO.File_Type;
        subtype Mode is Text_IO.File_Mode;
end Simple_IO;

with Simple_IO;
use  Simple_IO;
procedure Demo
is
        My_File: File_Type;
begin
        Create                                  -- Create not visible
                (File           => My_File,
                 Mode  => Out_File,  -- Out_File not visible
                 Name  => "XYZZY");
end Demo;
```

The lack of this capability makes it impossible to re-export a type from a package together with all the operations on that type.

This issue was introduced at the January 1987 ALIWG meeting, and was approved at the August 1987

meeting by a vote of 11 to 0 (with 10 abstaining). An extensive discussion of related issues may be found in "Composable Ada Software Components and the Re-Export Paradigm," Bryce Bardin and Christopher Thompson, Ada Letters, Jan 1988.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

1. Use derived types instead of subtypes to effect renaming. However, this introduces an undesirable type incompatibility which can only be overcome by frequent type conversions.

2. Individually rename each operation and each value (as necessary) at the same place that subtype is defined. This technique results in unnecessary work on the part of the Ada programmer (introducing the potential for error, e.g., incorrect renaming). Unfortunately, there is no way to re-export enumeration literals without losing either the ability to overload them or their static-ness.

POSSIBLE SOLUTIONS:

(1) Add type renaming to the language.

(a) Add appropriate syntax and semantics to Section 8.5, Renaming Declaration.

Possible Syntax:

```
renaming_declaration ::=
    ...
    | type identifier renames type_name;
```

Possible semantics (in paragraph form similar to the descriptions of the semantics for other renaming declarations - see LRM 8.5 paragraphs 4-7):

The X form of renaming declaration is used for the renaming of types. The determination of the entity in the elaboration of a renaming of a type involves identification of the set of values and the set of operations which characterize the renamed type. At any point where the renaming declaration is visible the set of values and the set of operations of the type are also visible.

The set of operations which characterize the renamed type are the basic operations or that type and those subprograms of that type which are derivable according to paragraph 3.4(11).

For each operation of the renamed type a subprogram renaming declaration is implicitly performed at the place of the type renaming declaration, where the subprogram_specification in the renaming declaration is identical to the subprogram_specification of the operation being renamed, under the following conditions:

1. Subprograms which are derivable are implicitly renamed after basic operations are renamed.

2. The implicit renaming of an operation is not performed if a declaration which is homographic to the operation is declared, renamed (other than basic operations), or instantiated immediately within the same declarative region as the renaming declaration.

If the renamed type is an enumerated type then for each enumeration literal of the renamed type a subprogram renaming declaration is implicitly performed at the place of the type renaming declaration, where the subprogram_specification in the renaming declaration is identical to the subprogram_specification of the enumeration literal being renamed, under the following conditions:

1. The implicit renaming of an enumeration literal is not performed if a declaration which is homographic to the enumeration literal is declared, renamed, or instantiated immediately within the same declarative region as the renaming declaration.

Note: This "homographic declaration" condition prevents multiple renames of a subprogram where the subprogram has 2 or more parameters of different types, and 2 or more of the types used in the subprogram are renamed in the same declarative region (This is a variation of the issue discussed in ALIWG Language Issue #42, "Provide deriving declaration to explicitly derive subprograms" which discusses the problem of derived subprograms from multiple derived types).

Also note: It would be desirable to preserve the static character of such renamed enumeration literals.

Example of the effect of this addition:

```
package Spatial_Math
is
    type Matrix is array (Integer range <>, Integer range <>) of Integer;
    type Vector is array (Integer range <>) of Integer;

    function "*" (V: Vector; M: Matrix) return Vector;
end Spatial_Math;

with Spatial_Math;
package Quarternion
is
    type Quat renames Spatial_Math.Vector (0 .. 3);
        -- Implicit renaming of Spatial_Math."*" and all array
        -- operations for the type Spatial_Math.Vector,
        -- e.g.
        -- function "*" (V: Vector; M: Matrix) return Vector
        --                                renames Spatial_Math."*";

    type Rot_Quat renames Spatial_Math.Matrix (0 ..3, 0 .. 3);
        -- No implicitly renaming of Spatial_Math."*" since
        -- that would create a homographic declarations declared
        -- immediately within the same declarative region
        -- which is illegal. Besides, the function has already been
        -- renamed in this declarative region

end Quarternion;


with Quarternion;
use Quarternion;
package Main
is
```

```
   Q: Quarternion.Quat              := (1, 2, 3);
   R: Quarternion.Rot_Quat    := ((1, 2, 3), (1, 2, 3), (1, 2, 3));

begin

      Q := Q * R;
      -- Call to Quarternion."*" (Q, R) which is really a
      -- call to Spatial_Math."*" (Q, R)
end Main;
```

b. Delete 8.5(16).

c. Also need to modify 7.4.1(1) to read:

If a private type declarations is given ... The corresponding declaration must be either a full type declaration[, a type renaming declaration,] or the declaration of a task type....

2. An alternate solution is define the concept of operator inheritance that appears in object-oriented languages.

PROVIDE GENERIC FORMAL EXCEPTIONS (SIGADA ALIWG LANGUAGE ISSUE 39)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

There is no way to specify the handling of exceptions raised by an actual subprogram parameter in a generic unit unless the exceptions are global to both the generic body and the unit the performs the instantiation.

For example:

```
generic
        with procedure Formal_Procedure (...);
procedure Generic_Do_Something (...);

procedure Generic_Do_Something (...) is
begin
        Formal_Procedure (...);
                -- What exceptions does the actual procedure raise?
exception
        when Constraint_Error =>
                ...
        when others =>
-- only handler for all exceptions raised by the actual procedure
-- corresponding to Formal_Procedure, unless they are
-- global (e.g., Constraint_Error). Can not do much except clean
-- up and re-propagate the exception.
                ...
end Generic_Do_Something;
```

This proposal was discussed at the August 1987 ALIWG meeting where it received unanimous support.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Define a separate package containing all the exceptions that the generic can handle by name. This package would be with'ed by both the generic and the subprogram to be used as the actual parameter in the generic instantiation. The actual generic subprogram parameter can only raise exceptions visible to both units and predefined exceptions. However, this technique involves defining the exceptions that are to be propagated into the generic prior to defining the generic rather than at the point where the generic is instantiated.

POSSIBLE SOLUTIONS:

1. Add generic exception parameters. The semantics of the association of generic formal exception parameters with actual exception parameters would be the same as exception renaming. The architect of the generic would specify the exception parameters to the generic and in comment form, list the exceptions which should be propagated for each generic subprogram formal parameter.

   For example:

```
generic
        Bad_Parameter_1_Error:      exception;
        Bad_Parameter_2_Error:      exception;
        Cannot_Locate_Error:  exception;
        ...

with procedure Formal_Procedure (...);
        --
        -- The following exceptions should be raised by
        -- Formal_Procedure in order to maximize the error recovery
        -- performed by Generic_Do_Something
        --
        -- Bad_Parameter_1_Error      If the 1st parameter is illegal
        --
        -- Bad_Parameter_2_Error      If the 2nd parameter is illegal
        --
        -- Cannot_Locate_Error        If ....
        --

procedure Generic_Do_Something (...);


procedure Generic_Do_Something (...) is
begin
        Formal_Procedure (...);
                -- What exceptions does the actual procedure raise?
exception
        when Bad_Parameter_1_Error   =>
                ...
        when Bad_Parameter_2_Error   =>
                ...
        when Cannot_Locate_Error     =>
                ...
                raise;  -- Propagate the actual exception associated
```

```
                    -- with Cannot_Locate_Error
        when Constraint_Error =>
                ...
        when others     =>
                ...
    end Generic_Do_Something;
```

2. In addition (or instead of - you do not really need both) to the generic exception parameter mechanism, define "exception parameters" for subprograms.

   The generic exception parameter mechanism suffers from the fact that there is no way formally defining the exceptions which are propagated by a subprogram or an entry. As a result, the software architect still cannot be certain that the user of the generic provided an actual subprogram parameter which propagated the actual generic exception parameters. Therefore the generic exception parameter mechanism is only a partial solution to this problem (see separate 9X Revision Request).

DEFINE ARGUMENT IDENTIFIERS FOR LANGUAGE-DEFINED PRAGMAS (ALIWG LANGUAGE ISSUE 64)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

The syntax of pragma's [LRM 2.8(2)] indicates that name association can be used in the argument_association when the user specifies a pragma. However, the LRM does not define the names of most of the parameters for any of the pragma's with the exception of the "ON" parameter for pragma Suppress (Section 11.7).

This issue was discussed and approved unanimously at the August ALIWG meeting.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Add names of parameters to the definitions of the language-defined pragmas. At this stage, adding names may require a consensus of the compiler developers.

ALTERNATE WAYS TO FURNISH A SUBPROGRAM BODY (SIGADA ALIWG LANGUAGE ISSUE 32)

DATE: July 22, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783
E-mail: hermix!colbert@rand.org

ANSI/MIL-STD-1815A REFERENCE: 6.3, 8.5, 12.3

PROBLEM:

According to the LRM: "A subprogram body specifies the execution of a subprogram" [LRM 6.3(1)].

"...For each subprogram declaration, there must be a corresponding body (except for a subprogram written in another language, as explained in section 13.9)."
[LRM 6.3(3)]

"A renaming declaration declares another name for an entity" [LRM 8.5(1)]

"An instance of a generic unit is declared by a generic instantiation."
[LRM 12.3(1)]

However, in situations where its desirable to hide implementation details, it is often desirable to declare subprogram in the visible part of a package and specify that the execution of the "declared" subprogram is an defined by the instance of a generic or renaming of a subprogram or entry.

This language issue was first discussed at the January 1987 meeting of the Ada Language Issues Working Group, where it received support by a large majority.

Example:

```
package List is
        type List_Type is limited private;
        ...
        procedure Sort (List:  in out List_Type);
        ...
        pragma Inline (Sort);
end;

with Quick_Sort;
package body List is
```

```
        procedure Quick_Sort_List is
                        new Quick_Sort (Sort_Type => List_Type);

        procedure Sort (List:  in out List_Type) is
        begin
                Quick_Sort_List (List);
        end;
        ...
end;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

1. Instantiate the generic (or rename existing subprogram) in the package specification.

```
with Quick_Sort;
package List
is
        type List_Type is limited private;
        ...
        procedure Sort is
                        new Quick_Sort (Sort_Type => List_Type);
        ...
end;
```

   This technique results in additional compilation dependencies, since any compilation unit that "with's" in the package now indirectly depends on the generic unit (or the unit containing the procedure to be renamed). This technique may also result in making other declarations visible which would otherwise be hidden. Finally, in some cases it is not possible to perform the instantiation in the package specification (e.g., if List_Type had a discriminant that did not have a default [LRM 12.3.2(4)]).

2. Declare a subprogram in the package declaration and call the instantiation of the generic or the renamed subprogram in the package [see above]. This technique results in unnecessary work on the part of the Ada programmer (introducing the potential for error). It also results in the declaration of procedures and functions which are otherwise unnecessary. To avoid additional run-time overhead, the exported function must have a Pragma Inline, which creates undesirable compilation dependencies on the package body.

POSSIBLE SOLUTIONS:

Change LRM 6.3(3) to read:

"...For each subprogram declaration, there must be a corresponding body ["body instantiation or body renames"] (except for a subprogram written in another language, as explained in section 13.9)"

Add appropriate syntax and semantics to Section 8.5 Renaming Declarations, and Section 12.3 Generic Instantiation.

Possible Syntax:

```
renaming_declaration ::=
        ...
        | subprogram_specification body renames
                                                subprogram_or_entry_name

generic_instantiation ::=
        ...
        | procedure identifier body is new
                        generic_procedure_name [generic_actual_part];
        | function identifier body is new
                        generic_function_name [generic_actual_part];
```

Note that care must be taken to make circularities of definition illegal:

```
package Q is
        procedure P;
        procedure R renames P;
end Q;

package body Q is
        procedure P body renames R;  -- would have to be illegal
end Q;
```

It was pointed out at the August 1989 meeting that from a minimalistic point of view, it would be sufficient to have only the capability of furnishing a body by renaming. Furnishing an instantiation could, in this case, still be achieved by using the renaming mechanism on a local instantiation within the body.

CONFORMANCE RULES SHOULD BE SIMPLIFIED

DATE: September 14, 1989

NAME: Randall Brukardt & Daniel Stock

ADDRESS: R.R. Software, Inc.
P.O. Box 1512
Madison WI 53704

TELEPHONE: (608) 244-6436

ANSI/MIL-STD-1815A REFERENCE: 6.3.1

PROBLEM:

The current rules for conformance are both confusing and more difficult than necessary to implement.

IMPORTANCE: ADMINISTRATIVE

The current rules for conformance appear arbitrary to users, while complicating life for compiler writers. Keeping the current arcane rules would continue to make Ada unnecessarily difficult to learn.

CURRENT WORKAROUNDS:

None. The current rules must be followed. Even worse, they must be learned - or ignored and left as a yet another resort reserved for Ada gurus only. The latter approach appears to be more common, giving users another reason to feel a little uncomfortable with Ada.

POSSIBLE SOLUTIONS:

Replace 6.3.1(3) and 6.3.1(4) with a statement that corresponding names in conforming declarations may be syntactically different as long as the names denote the same entity. This is simpler to understand than the current rule with its ramifications (such as AI-350 and AI-493). It is also simpler to implement, as a compiler need not keep track of which form of a name was used in a subprogram declaration.

Also, allow the reserved word "IN" to be ignored in conformance, unless it is followed by the reserved word "OUT". Similarly, allow sequences of (possibly multiple) parameter specifications to conform if they are equivalent to conforming sequences of single parameter specifications, so that conformance would no longer appear to be an exception to the general equivalence rules given in 3.2(10-11). One effect of these two changes is to permit conformance in the examples in note 6.3.1(8).

Note that both of these solutions are backward compatible.

CONFORMANCE RULE CONSISTENCY

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 6.3.1 (6.7, 8.5)

PROBLEM:

The conformance rules are inconsistent in their strictness. Most parts conform if they are syntactically equivalent while other parts must be identically parsed. (compare 6.3.1-7 and 6.3.1-8)

IMPORTANCE: ADMINISTRATIVE (for consistency)

CURRENT WORKAROUNDS:

The ACVC has removed the tests on some of the abnormally strict aspects of conformance, and many of the compiler implementors are adhering to the ACVC rather that 1851A.

POSSIBLE SOLUTIONS:

Allow a parameter specification with the default "in" mode to conform to a parameter specification with the explicit "in" mode. Allow a single parameter specification with an identifier list having multiple identifiers to conform to multiple parameter specification with an identifier lists having single identifiers with conforming names in the same order with the same conforming type. (This should be reflected in 6.7 and 8.5.)

NAMING OF THE SUBPROGRAMS TO WHICH AN INCLINE PRAGMA APPLIES

DATE: October 11, 1989

NAME: Mats Weber
Endorsed by Ada-Europe, Number AE-017,
originator : Stef Van Vlierberghe

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail : madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 6.3.2

PROBLEM:

In Ada 83 it is not possible to write a pragma Inline that applies only to a subset of overloaded subprograms in a declarative part. Moreover, the rules that state exactly to which subprograms a pragma Inline applies are overly complicated (especially in the presence of implicitly declared subprograms).
Example:

```
package Modulo_Arithmetic is

  type Int is range 0..12;

   -- function "*" (Left, Right : Int) return Int;   --(3)
   -- is implicitly declared here

   type Number is private;

   function "*" (Left, Right : Number) return Number;  --(1)

   function "*" (Left : Int; Right : Number) return Number; --(2)
private

  type Number is new Int;

   pragma Inline("*");
   -- It is impossible to write a pragma inline that only applies    -- to (1) but not (2).
   -- The existence of implicit declaration (3) creates ambiguities.

end Modulo_Arithmetic;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use renaming declarations for selective inlining. This creates additional names for the sole purpose of inlining.

POSSIBLE SOLUTIONS:

Introduce a notation that unambiguously identifies a given subprogram, for instance:

```
<subprogram_name>   [(<parameter_name> : <parameter_type>
                    {, <parameter_name> : <parameter_type>})]
```

or

```
<subprogram_name>   [(<parameter_type> {, <parameter_type>})]
```

the pragma in the above example would be written

```
pragma Inline("*"(Left, Right : Num));
```

or

```
pragma Inline("*"(Num, Num));
```

Note that this implies a change in the syntax of a pragma in LRM 2.8.

PRAGMA INLINE APPLIED TO A SUBPROGRAM RENAME

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 6.3.2

PROBLEM:

An INLINE pragma applied to a rename of a subprogram should only affect calls that use the rename (directly, or indirectly if the rename is itself later renamed). This allow useful constructs such as

```
package P is
        procedure Foo (X : Integer);

        procedure Inlined_Foo (X : Integer) renames Foo;
        pragma Inline (Inlined_Foo);
end P;
```

which allow the user to specify exactly which calls are to be inlined.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

INLINE SHOULD NOT APPLY TO ALL OVERLOADS

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 6.3.2

PROBLEM:

The current rule that pragma INLINE applies to all overloaded subprograms of the given name in the same declarative part is inconvenient and creates maintenance problems.

It is inconvenient, because

```
procedure foo(X: my_special_type); --- with a body of 1000 lines

procedure foo(X: integer); -- with a 1-line body
pragma INLINE(foo); --- OOPS ! gets the first foo, too
```

(To resort to renaming to restrict the applicability of the pragma is a horrible solution.)

It is problematic, because

the original code...

```
package bar is
        --- 500 lines of specifications....
        procedure foo(X: integer); -- with a 1-line body
        pragma INLINE(foo); --- (1)
end bar;
```

may have to be modified to become

```
package bar is
        procedure foo(X: my_special_type); --- with a body of 1000 lines
        --- 500 lines of specifications....
        procedure foo(X: integer); -- with a 1-line body
        pragma INLINE(foo);
end bar;
```

Unless one scans the entire specification for INLINE pragmas, there is the danger of unintentional inlining of newly introduced subprograms.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

6.3.2 (3), 2. sentence, should be rescinded. Pragma INLINE should apply only to the closest preceding subprogram of the specified name.

PROGRAM UNIT NAMES

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 6.3(2), 7.1(2), 9.1(3)

PROBLEM:

The language allows a program unit name to be appended to the "end" of a block or program unit. However, especially in blocks and program units with long declarative parts and/or exception handlers and packages with long private parts, it is advantageous to document the other main divisions of a block or program unit--"private", "begin", "exception". Currently, this must be done with comments. However, it would be better to allow the block or program unit name to be optionally appended to the other constructs as well.

Examples.

```
PACKAGE stack_handler IS
  -- declarations
PRIVATE stack_handler
  -- private part
END stack_handler;

PROCEDURE put (label : IN      string) IS
  -- declarations
BEGIN put
  -- sequence of statements
EXCEPTION put
  -- exception handlers
END put;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Comment the "private", "begin", and/or "exception" of the block or program unit with the name.

POSSIBLE SOLUTIONS:

GENERIC SUBPROGRAMS AND SUBPROGRAM BODIES

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 6.3(3)

PROBLEM:

Generic subprograms and subprogram bodies

> section:06.03(03)
> For each subprogram declaration, there must be a corresponding body...
>
> section12.03(05):
> the instance of a generic procedure is a procedure,#...

what is presumed to mean "the instance of a generic procedure is a procedure specification and a procedure body"

Subprogram bodies cannot be supplied by using a generic instantiation or a renaming declaration. As a result, parameter profiles need to be duplicated, and subprogram names need to be duplicated in generic instance names and pragma INLINE to obtain this effect. This duplication is a violation of the factorization principle. What would be the problem of allowing body supply by renaming or instantiation?

The intent of the example is to support varying strings with the same comparison operations as ada strings.

```
package VSTRING is
        type T is limited private;
        function "=" (LEFT,RIGHT:T) return BOOLEAN;
        function "<=" (LEFT,RIGHT:T) return BOOLEAN;
        function ">=" (LEFT,RIGHT:T) return BOOLEAN;
        function "<" (LEFT,RIGHT:T) return BOOLEAN;
        function ">" (LEFT,RIGHT:T) return BOOLEAN;

private
        type T is
        ...
end VSTRING;
```

If one tries to avoid duplication of relatively complex code, a generic seems to be appropriate.

```
package body VSTRING is
        generic
                with function COMPARE(LEFT,RIGHT:CHARACTER) return BOOLEAN;
        function LEXICOGRAPHIC(LEFT,RIGHT:T) return BOOLEAN;
        function LEXICOGRAPHIC(LEFT,RIGHT:T) return BOOLEAN is
        begin
        ---Do lexicographic comparison
        ...
        end LEXICOGRAPHIC;

        function EQUAL is new LEXICOGRAPHIC (COMPARE=>"=");
        pragma INLINE(EQUAL):
        function "=" (LEFT, RIGHT:T) return BOOLEAN is
        begin
                return EQUAL (LEFT, RIGHT);
        end "=";
        function LESS_OR_EQUAL is new ...
        ...
        end VSTRING;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

The suggested alternative is a change of the definition of the instantiation, that could be changed to
section 12.03(05):
The instance of a generic procedure is a procedure body,.

Currently the instantiation is illegal if at the place of the instantiation, a subprogram with the same name and parameter and result type profile would be visible in the same declarative region. So, instantiations, as currently defined, would keep their meaning as a result of:
section:06.03(03)
In absence of a subprogram declaration, the subprogram specification of the subprogram body acts as the declaration.

As a result this modification would be upward compatible.

Having the body-only semantics is needed in all cases were a generic instantiation or renaming declaration is used purely to get a suitable implementation for something announced in a specification.

If the rule would be changed, one would be able to write:

```
package body VSTRING is
        generic
                with function COMPARE(LEFT,RIGHT:CHARACTER) return BOOLEAN;
        function LEXICOGRAPHIC(LEFT,RIGHT:T) return BOOLEAN;
        function LEXICOGRAPHIC(LEFT,RIGHT:T) return BOOLEAN is
        begin
        ---Do lexicographic comparison
```

```
        ...
        end LEXICOGRAPHIC;
        function "=" is new LEXIGOGRAPHIC (COMPARE=>"=");
        --Furnished the body for the specification previously announced
        function "<=" is new...
        function SPECIAL is new LEXICOGRAPHIC (COMPARE => MY_COMPARE);
        --No specification was given in the declarative region, so the header of the instantiated body server
        as specification, as before.
        ...
end VSTRING;
```

Since the former solution contains at least 4 references to each function declaration, while the latter only contains 1, one might conclude that the factorization of the latter implementation is better. Since it is widely accepted that good factorization has a positive influence on reliability, ease on maintenance, and readability, one might prefer the latter solution.

One might say that the statements about the design goals and sources also apply here:

> section 01.03(03) : .. we have tried to provide language constructs that correspond intuitively to what the users will normally expect.

Experience with Ada education learns that people expect the latter solution to work.

GENERIC SUBPROGRAMS AND SUBPROGRAM BODIES

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 6.3(3)

PROBLEM:

Generic subprograms and subprogram bodies

section :6.3(3)

For each subprogram declaration, there must be a corresponding body ...

section 12.3(5)

the instance of a generic procedure is a procedure, ...

what is presumed to mean "the instance of a generic procedure is a procedure specification and a procedure body"

Subprogram bodies cannot be supplied by using a generic instantiation or a renaming declaration. As a result, parameter profiles need to be duplicated, and subprogram names need to be duplicated in generic instance names and pragma INLINE to obtain this effect. This duplication is a violation of the factorization principle. What would be the problem of allowing body supply by renaming or instantiation?

The intent of the example is to support varying strings with the same comparison operations as ada strings.

```
package VSTRING is

    type T is limited private;

    function "="  (LEFT,RIGHT:T) return BOOLEAN;
    function "<=" (LEFT,RIGHT:T) return BOOLEAN;
    function ">=" (LEFT,RIGHT:T) return BOOLEAN;
    function "<=" (LEFT,RIGHT:T) return BOOLEAN;
    function ">=" (LEFT,RIGHT:T) return BOOLEAN;

Private

    type T is
```

```
        ...
end VSTRING;
```

If one tries to avoid duplication of relatively complex code, a generic seems to be appropriate.

```
package body VSTRING is

        generic
                with function COMPARE (LEFT, RIGHT:CHARACTER) return
BOOLEAN;

        function LEXICOGRAPHIC(LEFT,RIGHT:T) return BOOLEAN
        function LEXICOGRAPHIC(LEFT,RIGHT:T) return BOOLEAN is
        begin
        -- Do lexicographic comparison
        ...
        end LEXICOGRAPHIC;
        function EQUAL is new LEXICOGRAPHIC (COMPARE =>"=");
        pragma INLINE(EQUAL);
        function "=" (LEFT,RIGHT:T) return BOOLEAN is
        begin
                return EQUAL (LEFT,RIGHT);
        end "=";

        function LESS_OR_EQUAL is new ...
        ...
end VSTRING;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The suggested alternative is a change of the definition of the instantiation that could be changed to

section 12.3(5):

The instance of a generic procedure is a procedure body, ...

Currently the instantiation is illegal if at the place of the instantiation, a subprogrammer with the same name and parameter and result type profile would be visible in the same declarative region. So instantiations, as currently defined, would keep their meaning as a result of:

section 6.3(3):

In absence of a subprogram declaration, the subprogram specification of the subprogram body acts as the declaration.

As a result this modification would be upward compatible.

Having the body--only semantics is needed in all cases were a generic instantiation or renaming declaration is used purely to get a suitable implementation for something announced in a specification.

If the rule would be changed, one would be able to write:

```
package body VSTRING is

        generic
                with function COMPARE(LEFT,RIGHT:CHARACTER) return
BOOLEAN;
        function LEXICOGRAPHIC (LEFT,RIGHT:T) return BOOLEAN;

        function LEXICOGRAPHIC (LEFT,RIGHT:T) return BOOLEAN is
        begin
        -- Do lexicographic comparison
        ...
        end LEXICOGRAPHIC;

        function "=" is new LEXICOGRAPHIC (COMPARE => "="(L
        --      Furnishes the body for the specification previously announced
        function "<=" is new ...

                function SPECIAL is new LEXICOGRAPHIC (COMPARE => MY_COMPARE);
        --      No specification was given in the declarative region, so the
        --      header of the instantiated body server as specification, as
        --      before.
        ...
end VSTRING;
```

Since the former solution contains at least 4 references to each function declaration, while the latter only contains 1, one might conclude that the factorization of the latter implementation is better. Since it is widely accepted that good factorization has a positive influence on reliability, ease of maintenance, and readability, one might prefer the latter solution.

One might say that the statements about design goals and sources also apply here:

> Section 1.3(3): ... We have tried to provide language constructs that correspond intuitively to what the users will normally expect.

Experience with Ada education learns that people expect the latter solution to work.

RENAMING DECLARATIONS AS SUBPROGRAM BODIES

DATE: July 21, 19893

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of Ada UK.

ADDRESS: IPSYS plc
Marlborough Court
Pickford Street
Macclesfield
Cheshire SK11 6JD
United Kingdom

TELEPHONE: +44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 6.3, 12.3

PROBLEM:

It is not possible to separate the subprogram specification and body where the implementation is to be a renaming declaration. This is a particularly common requirement when mapping from some high-level component-based design description to Ada.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

There are two workarounds. The first is to use the renaming declaration in the package specification directly. For example:

```
with LOW_LEVEL_OBJECT;

package HIGH_LEVEL-OBJECT is
        procedure HIGH_LEVEL_OPERATION
                renames LOW_LEVEL_OBJECT.LOW_LEVEL_OPERATION;
end HIGH_LEVEL_OBJECT;
```

This however violates the principle of separation of specification and implementation by making the implementation choice for the subprogram directly visible, and results in a stronger specification dependency on the package containing the renamed or instantiated declaration.

The second possibility is to retain the separate subprogram specification and body, and provide a call to the required subprogram. For example:

```
package HIGH_LEVEL_OBJECT is
   procedure HIGH_LEVEL_OPERATION;
end HIGH_LEVEL_OBJECT;
```

```
with LOW_LEVEL_OBJECT;
package body HIGH_LEVEL_OBJECT is
   procedure HIGH_LEVEL_OPERATION is
   begin
     LOW_LEVEL_OBJECT.LOW_LEVEL_OPERATION;
   end HIGH_LEVEL_OPERATION;
end HIGH_LEVEL_OBJECT;
```

Where the Ada is being generated automatically this approach requires an analysis of the subprogram specification in order to generate the required actual parameter list.

POSSIBLE SOLUTIONS:

Allow a subprogram renaming declaration or subprogram instantiation as a subprogram body.

As an example:

```
package HIGH_LEVEL_OBJECT is
   procedure HIGH_LEVEL_OPERATION;
end HIGH_LEVEL_OBJECT;

with LOW_LEVEL_OBJECT;
package body HIGH_LEVEL_OBJECT is
   procedure HIGH_LEVEL_OPERATION
      renames LOW_LEVEL_OBJECT.LOW_LEVEL_OPERATION;
end HIGH_LEVEL_OBJECT;
```

INFIX FUNCTION CALL

DATE: October 24, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 6.4, 2.2

PROBLEM:

Readability of Ada code is often clumsy because the only binary operators that can be overloaded are predefined. Many binary operations have names that could be much more understandable if they were written out.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Use predefined binary operators:
or
Use function names and clump all the parameters together after the name

POSSIBLE SOLUTIONS:

Split the actual parameter part of a function call into a former actual parameter part and a latter actual parameter part. New delimiters, ")_" and "_(", could be added to help designate the function name in the construct when the infix form is used. All rules for the actual parameter part should remain the same for the concatenation of the two actual parameter parts. The function declaration should not change.

Example:

```
Angular_Momentum := ( Radius )_Cross_( Velocity );
```

DISTINGUISHING PARAMETER MODES ON CALLS

DATE: February 17, 1989

NAME: Keith Thompson

ADDRESS: TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9891

TELEPHONE: (619) 457-2700 x121

ANSI/MIL-STD-1815A REFERENCE: 6.4.1

PROBLEM:

Given a procedure call, there is no way to tell the mode (IN, OUT, or IN OUT) of the actual parameters. This can make it impossible to determine whether a given variable may be modified by a call without reference to the specification of the procedure.

IMPORTANCE: IMPORTANT

Consequences if this request is not satisfied: it will continue to be difficult to determine by visual inspection whether objects are modified by procedure calls.

CURRENT WORKAROUNDS:

The information can be provided via comments on the call.

For example:

```
PROC( FORMAL_1 => ACTUAL_1,  -- in
      FORMAL_2 => ACTUAL_2,  -- in out
      FORMAL_3 => ACTUAL_3 );-- out
```

This is error-prone, since there is no check that the comments are correct.

IN mode parameters can be indicated by parentheses, but this doesn't work for IN OUT or OUT parameters. Programming support tools can be used to get the information from the procedure specification, but such tools are not always available (particularly when the software is published in a book or periodical).

POSSIBLE SOLUTIONS:

Allow optional specification of parameter modes on procedure calls. The mode is specified by preceding the parameter association with IN, OUT, or IN OUT. This is allowed for both positional and named associations. The mode specified for a parameter association must match the mode of the actual parameter; otherwise, the association is illegal. The mode IN may be specified for an association whether the

corresponding formal parameter is explicitly or implicitly of mode IN.

Example:

```
PROC( in        FORMAL_1 => ACTUAL_1,
      in out    FORMAL_2 => ACTUAL_2,
      out       FORMAL_3 => ACTUAL_3 );
```

For symmetry, this should be allowed for function calls as well as procedure calls, even though the only legal mode is IN. It should also be allowed for generic parameter associations corresponding to generic formal objects.

The current grammar for parameter_association is:

```
parameter_association ::=
        [formal_parameter =>] actual_parameter
```

The revised grammar would be:

```
actual_mode ::= in | in out | out

formal_parameter ::=
        [actual_mode] [formal_parameter =>] actual_parameter
```

where

```
mode ::= [in] | in out | out
```

Note that this change has no impact on current software, since the mode specification is optional and no new reserved words are introduced. It should be a simple and straightforward change in most or all compilers.

DEFAULT VALUES FOR SUBPROGRAM PARAMETERS MAY REFERENCE OTHER PARAMETERS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 6.4.2

PROBLEM:

Many times it would be convenient to have a subprogram parameter default to an expression involving earlier (in mode) parameters:

```
function Enclosing_Directory (File : String) return String;

procedure P (File : String;
             Directory : String := Enclosing_Directory (File));
```

When a call uses the default values in such a case, an evaluation order is imposed on the parameters (the order in which the formal parameters are declared is always a valid evaluation order). This evaluation order should be imposed only at call sites in which actual parameters have been defaulted to expressions involving prior parameters. Forcing this order to be used for all call sites might be disastrous when coding calls to predefined operators, for example.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Force all call sites to specify all parameters (somewhat error prone and less maintainable) or introduce overloaded versions of the subprogram (less readable, since it is not clear from the specification what the 'default' values for the missing parameters are).

POSSIBLE SOLUTIONS:

IMPROVED OVERLOAD RESOLUTION

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 6.6(6)

PROBLEM:

Overload resolution in Ada is weaker than it has to be; there are many cases in which stronger analysis would be able to disambiguate overloading that is not compilable under the existing standard.

IMPORTANCE: IMPORTANT

Overloading promotes object-oriented programming and should be supported by the standard in as many ways as possible.

CURRENT WORKAROUNDS:

Live within the current constraints on overloading.

POSSIBLE SOLUTIONS:

Extend the range of "clues" used in overload resolution to include parameter names, subtypes (perhaps), and other kinds of information currently no used for disambiguation. For example:

--These two functions are NOT homographs, because the --parameter names are different:

```
function Is_Valid (This : in String) return Boolean;
function Is_Valid (That : in String) return Boolean;

if Is_Valid (This => Foo) then...
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

OVERLOADING OF EXPLICIT EQUALITY "="

DATE: September 27, 1989

NAME: K. Buehrer/ESI

ADDRESS: Contraves AG
8052 Zuerich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 6.7, 4.5.2

PROBLEM:

Explicit definition of the equality operator "=" is only allowed for limited types. Moreover, both operands must be of the same limited type. This is clearly insufficient in many instances, e.g:

- Rational numbers: The same value may be represented in several different ways.
- Private types implemented as access types: Equality should address the designated objects, not the access objects.
- Character string types implemented as discriminant records: Equality should compare the actual length but not the maximal length, and comparison with predefined type string should be possible.

Overloading is needed for the equality operator "=" as for all the other predefined relational operators.

IMPORTANCE: ESSENTIAL

The expressive strength of the Ada language is reduced by the absence of overloading for "=". All rational operators may be overloaded except equality. This is a strange singularity of the language.

CURRENT WORKAROUNDS: NONE

There is no real workaround The only thing to do is to declare a function like:

```
FUNCTION equal (left : some_type;
                right : any_type) RETURN boolean;
```

or to make the type limited private. However, both possibilities are insufficient. Defining a function "equal" does not prohibit the use o predefined equality, returning an incorrect result. Making a type limited private has other drawbacks: Equality can only be defined for two operands of the same limited type, many useful constructs are not anymore available (":=", out parameters, aggregates, object initialization) and all composite types having components of the limited type become limited as well.

POSSIBLE SOLUTIONS:

Declaration of a function to overload "=" is allowed for any type, not just for limited types:

```
FUNCTION "="  (left : any_type;
               right : any_type) RETURN boolean;
```

Of course, as for the other relational operators, overloading of equality would only affect explicit use of "=", but no basic comparisons (see 6.7(6). Similarly, overloading of equality would not extend to composite types (see 4.5.2.(8.9). This in mind, there is absolutely no reason why we should treat equality and other relational operators differently. Why should overloading, considered a virtue of Ada operators in general, be so vicious when applied to equality?

The suggested extension is strictly upward compatible. No negative impact on existing implementations is to be expected.

OVERLOADING OF EXPLICIT ASSIGNMENT ":="

DATE: September 27, 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraves AG
8025 Zuerich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 6.7, 5.2, 8.7

PROBLEM:

Explicit overloading (or redefinition) of the assignment operation ":=" is not supported by the language. However, this feature would be very useful in several instances (compare request for overloading of "=") e.g:

- Rational numbers: Assignment may include normalization.
- Private types implemented as access types: Assignment may be required to address the designated values, not just the pointers.
- Character string types implemented as discriminant records: Assignment between constrained objects of different maximal length and assignment of values of predefined string must be possible.

IMPORTANCE: IMPORTANT

Overloading of the assignment operation would clearly increase the expressive strength of the language and fit nicely into its overall philosophy. The available workaround is clumsy and unsafe.

CURRENT WORKAROUNDS:

Declaration of a procedure like:

```
PROCEDURE set    (object : OUT some_type;
                  to     : some_type);

PROCEDURE assign (value  : IN some_type;
                  to     : OUT some_type);
```

However, there is no way to prohibit the use of predefined assignment, producing incorrect results.

POSSIBLE SOLUTIONS:

Declaration of a procedure to overload ":=" is allowed for any type:

```
PROCEDURE ":="    (left : OUT any_type;
                   right : IN any_type);
```

The mode IN OUT cannot be allowed for the left parameter in general, because Ada requires copy semantics for scalar types. Copy-in of undefined scalar values will raise exceptions at run time. The bounds (array types) or discriminants (record types) of the left operand are nevertheless available. For access types however, the mode IN OUT would sometimes be more convenient, e.g. to allow unchecked deallocation of the previously allocated object.

The suggested extension is strictly upward compatible. NO negative impact on existing implementations is to be expected, overloading resolution has to be applied to assignment already today.

ADDING CONSTRUCTORS AND DESTRUCTORS TO PACKAGE TYPES

DATE: August 18, 1989

NAME: Goran Karlsson

ADDRESS: Bofors Electronic AB
Nettovagen 6
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 222 90

ANSI/MIL-STD-1815A REFERENCE: 6.7 (possibly)

PROBLEM:

Garbage collection and others

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

This idea has been fetched from the C++ language in which it is possible to declare specific constructors for a type.

A constructor is normally called when the data is declared. A constructor may also perform some kind of type conversion, although there is one that is used for initialization, copy in of function argument and return from function.

A destructor is called when a variable is deallocated, either explicitly or by exiting the scope of the variable.

It is sometimes necessary to reference the instance of the package type within procedures or functions in the body. One could either use the package type name or the reserved word ALL, both of which will contain the instance of the type on which the functions or procedures operates. In the example below ALL has been used.

Empty or default constructor

```
FUNCTION package_type(value : IN package type)
        RETURN package_type;
```
Call by declaring a variable of type type or by new type;

Initialization, copy in argument or return value:

```
FUNCTION package_type(value:IN package_type)
          RETURN package_type;
```

Construction or conversion:

```
FUNCTION package_type(any_IN_arguments) RETURN package_type;
Call by declaring a variable thus:
        variable:type(any_IN_arguments);
or:
        new type (any_IN_arguments);
or:
        variable:=type(any_IN_arguments);
```

Destructor:

```
PROCEDURE package_type();
Call destructor by:
      delete value;
```

Here follows an example of a string data type using constructors and destructors. It uses reference counting in order to perform garbage collection. The program is a translation of an example in the C++ programming language by Bjarne Stroustrup in chapter 6.9 page 184. (Addison and Wesley ISBN 0-201-12078-X)

```
PACKAGE Example IS

    Type new_string IS LIMITED PRIVATE;

    PACKAGE TYPE my_string_type IS

        p:  new string;

        -- Empty Constructor
        FUNCTION my_string_type()
                RETURN my_string_type;

        -- Constructor used for initialization arguments and
        -- return
        FUNCTION my_string_type(value:  IN my string_type)
                RETURN my_string_type;

        -- Conversion from normal string
        FUNCTION my_string_type(value:  IN string)
                RETURN my_string_type;

        -- Destructor
        PROCEDURE my_string_type();

        -- Indexing
        FUNCTION "()"(i:  IN POSITIVE) RETURN character;

        -- Some attributes implemented as functions
        -- (It would be nice with overloadable attributes!)
```

```
            FUNCTION length() RETURN INTEGER;

            FUNCTION first() RETURN INTEGER;

            FUNCTION last() RETURN INTEGER;

        END my string type;

        -- Assignment
        PROCEDURE ":="(destination:  OUT my_string_type;
                       data:  IN my_string_type);

PRIVATE

        TYPE string_array IS ARRAY(INTEGER range <>) OF character;

        TYPE ref IS ACCESS string_array;

        TYPE new_string_record IS RECORD
            ptr                 : ref;
            n                   :natural;
        END RECORD;

        TYPE new_string IS ACCESS new_string_record;

END Example;

PACKAGE BODY Example IS

        PACKAGE BODY my_string_type IS

            -- Empty Constructor
            FUNCTION my_string_type() IS
            BEGIN
                p := new new_string_record;
                p.ptr := null;
                p.n := 1;
                RETURN ALL;
            END my_string_type;

            -- Constructor used for initialization arguments and return
            FUNCTION my_string_type(value:  IN my_string_type) IS
            BEGIN
              value.p.n := value.p.n + 1;
              ALL.ALL := value.ALL; --Must avoid overloaded assignment!
              RETURN ALL;
            END my_string_type;

            -- Conversion from normal string
            FUNCTION my_string_type(value: IN string) IS
```

```
        BEGIN
                        p := new new_string_record;
                        p.ptr := new string_array(value'range);
                        p.ptr.ALL := value;
                        p.n := 1;
                        RETURN ALL;
                END my_string_type;

                -- Destructor
                PROCEDURE my_string_type();
                BEGIN
                        p.n := contents.p.n - 1;
                        IF p.n = 0 THEN
                                delete p.ptr;
                                delete p;
                        END if;
        END my_string_type;

        -- Indexing
        FUNCTION "()"(i: IN index) RETURN character;
        BEGIN
          RETURN p.ptr.(i);
        END "()";

        -- Some attributes
        FUNCTION length() RETURN natural IS
        BEGIN
          RETURN p.ptr.ALL'length;
        END length;

        FUNCTION first() RETURN INTEGER IS
        BEGIN
          RETURN p.ptr.ALL'first;
        END first;

        FUNCTION last() RETURN INTEGER IS
        BEGIN
          RETURN p.ptr.ALL'first;
        END last;

END my_string_type;

-- Assignment
PROCEDURE ":="(destination: OUT my_string_type;
               data: IN my_string_type);
BEGIN
  data.p.n. :=data.p.n. + 1;
  destination.p.n. := destination.p.n. - 1;
  IF destination.p.n. = 0  THEN
          delete destination.p.ptr;
          delete destination.p;
```

```
      END if;
      destination.p := data.p;
   END ":=";

   -- Comparison
   FUNCTION "="(s1,s2:IN my-string_type) RETURN boolean;
   BEGIN
      RETURN    s1.p.ptr.length = s2.p.ptr.length and then
                                  s1.p.ptr.characters(1..s1.p.ptr.length) =
                                  s2.p.ptr.characters(1..s2.p.ptr.length);
   END "=";

END Example;
```

ADDITIONAL OVERLOADED OPERATORS

DATE: August 18, 1989

NAME: Goran Karlsson

ADDRESS: Bofors Electronics AB
Nettovagen 6
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 222 90

ANSI/MIL-STD-1815A REFERENCE: 6.7

PROBLEM:

There are some "OBVIOUSLY" missing overloadable operators.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Using named procedures.

POSSIBLE SOLUTIONS:

It should be possible to overload the following operators:

For limited private types only:

```
PROCEDURE ":="(a:OUT any_limited_private?type;
                    b: IN any_type);
```

For all abstract indexable types such as list and others:

```
FUNCTION "()"(v: IN abstract_indexable_type)
                    RETURN any_type;
```

RE-DEFINE LIMITED PRIVATE

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 6.7

PROBLEM:

The present definitions of private and limited private demonstrate a confusion between assignment and comparison which is comparable to that of some older programming languages which used the same symbol for both. The desirability of hiding or replacing ":=", "=", and "/=" do not necessarily go hand in hand. When an access type is being defined, it may be correct in terms of the application domain that comparison is the comparison of the objects denoted, rather than the pointers. In this case, "=" must be replaced. This means the type is limited private, and assignment is not visible. If garbage collection is provided, or is not an issue, predefined assignment may be desired.

Although it is true that "=' and "/=" are generally opposite, it is also true that "<=" can generally be derived from "<" and "=", and that "<" and ">=" are generally opposite. This should be the concern of the type designer, rather than the language designer. The present status is inconsistent, since it singles out only equality and inequality.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use private types and do not use the incorrectly exported operations, or use limited private types and change the spelling of ":=" to "Copy" or "Assign". Define functions Eq and Neq when "/=" should not be the complement of "=".

POSSIBLE SOLUTIONS:

Eliminate private and limited private (or retain them only for compatibility), and introduce limited types, which behave like limited private types except that "=" and "/=" are decoupled. Permit the definition of ":=" as a procedure, as described in another revision request.

When all of the components of a record have equality defined (i.e., have "=" defined either implicitly of explicitly), the record should automatically have "=" defined, in terms of the definitions of "=" which are visible where the record type is declared.

OVERLOADING OF ASSIGNMENT

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 6.7

PROBLEM:

One of the design goals of the Ada programming language is extensibility. It should be, and largely is, possible to use Ada declarations to add things to the language which come to look like part of the language. One can, for example, define a numeric type which has properties exactly analogous to those of Standard.Integer, but which has limits appropriate to the problem at hand.

It is not possible, however, to create a non-numeric type which looks like a numeric type. A type resembling universal_integer can be created as pointers to arrays of integers. If such a type is private, assignment is restricted to an exact copy, which precludes garbage collection of the destination. If it is limited private, assignment is not available at all. Although the functionality can be provided by a procedure Assign, the syntax makes it clear that this type is second class; it does not look like a "genuine" Ada type.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

```
procedure Assign
        (Left: in out universal_integer;
         Right: in universal_integer);
```

POSSIBLE SOLUTIONS:

Permit assignment to be overloaded.

```
procedure ":="
        (Left: in out universal_integer;
         Right: in universal_integer);
```

or

```
procedure ":="
        (Left: out universal_integer;
         Right: in universal_integer);
```

Both forms should be allowed, since it may be desirable to avoid evaluating Left, or it may be desirable to finalize the value, particularly if universal_integer is an access type.

This overloading of ":=" for assignment suggests using the overloading where ":=" is used for initialization. Since there are already significant differences between these two uses of ":=" (e.g., sliding of arrays), either choice could be consistently made.

USER DEFINED OPERATORS

DATE: October 12, 1989

NAME: Craig Cowden

ADDRESS: Naval Security Group Detachment
Pensacola, Florida 32511

TELEPHONE: (904) 452-6399

ANSI/MIL-STD-1815A REFERENCE: 6.7

PROBLEM:

The lack of the ability to create user defined operators restricts the usage of infix operators to those generally associated with arithmetic. Typical logic programming operators, including "?" (query) and ":-" (if), are not predefined and not allowed. Therefore, users are constrained in their creativity to define intuitive operations.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The obvious workaround is to use functions in place of infix operators. This method often reduces readability of the code.

POSSIBLE SOLUTIONS:

Allow users to define their own infix operator using any ASCII character(s). The infix operator would be declared similarly to a function declaration except the operator would be enclosed in quotes. For example,

```
"?"(X : Logical_Clause) return Boolean;
```

would define the unary operator "?" (query). (A restriction to a set of characters and to the length of the operators may need to be considered for compiler efficiency.)

LIBERALIZATION OF OVERLOADING

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 6.7(1)

PROBLEM:

Currently, the language allows overloading of only logical, relational, binary adding, unary adding, multiplying, or highest precedence operators. The syntax of the language could be made more powerful by removing this restriction. At present, for example, it is inconvenient to write a package to handle logical connectives, since many of the symbols one would want to use (e.g. ->, <-, /, v, &, ^, <->) cannot be defined as overloaded operators.

Example:

```
FUNCTION "->"    (left  : IN    boolean;
                  right : IN    boolean)
                        RETURN boolean;
```

A related improvement would be to allow the overloading of a PROCEDURE ":=" to be used in the same way as an overloaded operator (viz., in infix notation). Such a procedure would be useful in packages declaring a limited private type, in which it is necessary to explicitly provide an assignment operation, or in a generic unit in which a generic formal type is limited private and an assignment operation needs to be imported. At present, one can provide such an operation, but one cannot name it ":=". It is more natural, however, to write x := y than assign(y, to => x).

Example:

```
PROCEDURE ":="   (item : in    items;
                  to   :   out items);
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Declare operators as ordinary functions and assignment as an ordinary procedure.

POSSIBLE SOLUTIONS:

For additional references to Section 6. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

0745 INTELLIGENT STRONG TYPING 3-67

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 7. PACKAGES

WHEN A PACKAGE PROVIDES A PRIVATE TYPE AND A NUMBER OF OPERATIONS ON THAT TYPE

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 7, 9

PROBLEM:

Currently, when a package provides a private type and a number of operations on that type, it has to be determined by the programmer whether or not the objects of that type are to be protected from simultaneous access from different tasks by using a task to contain each object. If the package is to be reusable, 2 versions of the package have to be written and maintained. Desirable to be able to write procedures to provide the operations, and have some way for the user of the package to be able to specify whether the procedures are to be transformed into accept statements (would need an extension to the syntax of a task entry to allow return values, if functions were also to be supported).

IMPORTANCE: ESSENTIAL

Ada will fail to become fully accepted as a language for writing modular software.

CURRENT WORKAROUNDS:

Use of source code preprocessors.

POSSIBLE SOLUTIONS:

ALLOW READ-ONLY ACCESS TO PACKAGE DATA OBJECTS

DATE: June 12, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 7

PROBLEM:

It is desirable to allow applications to specify that the data in a package should be read-only. Currently, the provision of a with clause permits generally unrestricted permission to both read and modify values of non-limited private, non-local variables. Time-sensitive applications would greatly benefit from a mechanism that would permit package data to be specified as read-only to some subprograms while read/write to others.

We note that this precise mechanism is available for data passed as parameters. We also realize that a portion of the requested capability is available by defining package data using private types and private sections, however, execution penalties to provide access and decomposition functions for these data structures in terms of both space and time are expected to continue to be prohibitive for time-sensitive applications.

Providing a read-only mechanism for package data would increase the level of protection the language provides for data security.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Applications are constrained to the current rules. Design and implementation standards have to consider that any data in a package is potentially writable by any program unit that references that package. Test planning is also impacted by this structure.

POSSIBLE SOLUTIONS:

We've been able to identify two potential enhancements to the language definition that would improve the situation. We admit to not having the resources to perform a full cost/benefit analysis on these solutions, but we are convinced that either strategy should have relatively small impact on compiler implementations.

The first possible solution is to extend the 'in' 'in out' parameter type marking to package names in with clauses. For example, one could write:

```
WITH IN package_1;
```

to specify that the compiler should treat the data in package_1 as read-only. We believe that modifications to current compilers to support this method should be relatively painless.

A second possible solution is to allow the provision of an optional list of names as part of the WITH clause. Access to resources defined in the package other than those called out in the list would be prohibited. This solution would be similar to that employed by the JOVIAL language; opportunities exist, though, to clean up the ambiguities present in the JOVIAL implementation.

SEMILIMITED TYPES

DATE: September 22, 1989

NAME: Lars Selsbo

ADDRESS: Contraves AG
Schaffhauserstrasse 580
CH-8052 ZURICH
Switzerland

TELEPHONE: +41 / 1 / 306 22 11

ANSI/MIL-STD-1815A REFERENCE: 7, 12, and 14.

PROBLEM:

It is a common situation that a choice has to be made between exporting a private type, where the predefined comparison for equality and inequality returns nonsense values, and exporting a limited type with a copy procedure, even though the implicitly declared assignment would have been sufficient.

To improve the data abstraction of the language, a "semilimited" type should be introduced, for which the assignment is implicitly declared, but for which the predefined comparison for equality and inequality is NOT implicitly declared.

(A type which works as a private type in respect to the predefined comparison for equality and inequality, and as a limited type in respect to implicit assignment, is also a semilimited type, but this case is of little practical importance.)

Example 1:

In Grady Booch's book "Software Components with Ada", a lot of bounded (Booch terminology) components could have exported semilimited types, instead of limited types.

If the generic formal types also were made semilimited instead of private, then it would be possible to create nested data structures of bounded components (maybe not that efficient).

```
generic
        type Item is semilimited private;
package Stack_Bounded is

        type Stack (The_Size : Positive) is semilimited private;

        procedure Push          (The _Item      : in Item;
                                 On_The_Stack   : in out Stack);

        procedure Pop (The Stack : in out Stack);

        function Top_Of (The_Stack : in Stack) return Item;
```

```
        generic
                function Is_Equal (     Left    : in Item;
                                        Right   : in Item) return Boolean;
        function Is_Equal (     Left    : in Stack;
                                Right   : in Stack) return Boolean;

        Overflow  : exception;
        Underflow : exception;

private

        type Items is array (Integer range <>) of Item;

        type Stack (The_Size : Positive) is
                record
                        The_Items : Items (1 .. The_Size);
                        The_Top : Natural := 0;
                end

        end Stack_Bounded;

package body Stack_Bounded is

        procedure Push (The_Item : in Item;
                        On_The_Stack : in out Stack) is


        begin
                If (On_The_Stack.The_Top = On_The_Stack.The_Size) then
                    raise Overflow;

                else
                    On_The_Stack.The_Top := On_The_Stack.The_Top + 1;
            On_The_Stack.The_Items (On_The_Stack.The_Top) :=
                            The_Item;

                end if;
        end Push;

        procedure Pop (The_Stack : in out Stack) is
        begin
                if (On_The_Stack.The_Top = 0) then
              raise Underflow;
                else
                 On_The_Stack.The_Top := On_The_Stack.The_Top - 1;
                end if;
        end Pop;

        function Top_Of (The_Stack : in Stack) return Item;
        begin
                if (On_The_Stack.The_Top = 0) then
```

```
            raise Underflow;
                else
                        return The_Stack.The_Items (The_Stack.The_Top);
                end if;
        end Top_Of;

        function Is_Equal (     Left    : in Stack;
                                Right   : in Stack) return Boolean is
        begin
            if (Left.The_Top /= Right.The_Top) then
                    return False;
             else
                    for Index in 1 .. Left.The_Top) loop
                        if (not Is_Equal (Left => Left.Items (Index)
                                Right => Right.Items (Index)) then

                     return False;
                        end if;
                   end loop;
                   return True;
                    end if;
        end Is_Equal;

end Stack_Bounded;
```

Example 2

Ada is often criticized because it is difficult to declare efficient variable length strings. The introduction of a semilimited private type would probably reduce this problem.

```
        package String_Utilities is

                type Bounded_String (The_Size : Positive) is semilimited private

                Copy ( From_The_String          :       in String;
                       To_The_Bounded_String    :       in out Bounded_String);

                Copy (From_The_Bounded_String : in Bounded_String;
              To_The_Bounded_String : in out Bounded_String);

                --
                -- Used to copy values between bounded strings with different
                -- discriminants.
                -- Maybe also more efficient than the implicit assignment.

                function String_Of
                        (The_Bounded_String : in Bounded String) return String;

                Overflow : exception;
```

```
private

        type Bounded_String (The_Size : Positive ) is
                record
                        The_String : String (1 .. The_Size);
                        The_Length : Natural := 0;
                end record;

end String_Utilities;

package body String_Utilities is

        Copy (From_The_String : in String;
         To_The_Bounded_String : in out Bounded_String) is

        begin
            if (From_The_String'Length >
       To_The_Bounded_String.The_Length) then
       raise Overflow;

    else
                    To_The_Bounded_String.The_Length :=
                        From_The_String'Length;
                        To_The_Bounded_String.The_Items
      (1 .. To_The_Bounded_String.The_Length := From_The_String;

                end if;
        end Copy;

        Copy (From_The_Bounded_String : in Bounded_String;
     To_The_Bounded String : in out Bounded_String) is

begin
                if (From_The_Bounded_String.The_Length >
         To_The_Bounded_String.The_Length) then
         raise Overflow;
         else
         To_The_Bounded_String.The_Length :=
                    From_The_Bounded_String.The_Length;
                  To_The_Bounded_String.The_Items
           (1 .. To_The_Bounded_String.The_Length) :=
          From_The_Bounded_String.The_Items
           (1 .. From_The_Bounded_String.The_Length);
        end it;
end Copy;

function String_Of
  (The_Bounded_String : in Bounded_String) return String is
begin
  return The_Bounded_String
       (1 ... The_Bounded_String.The_Length);
```

```
end;

private

type Bounded_String (The_Size : Positive) is
        record
                The_String : String (1 .. The_Size);
                The_Length : Natural := 0;
        end record;

end String_Utilities;
```

Notice in this example that both the implicitly defined assignment and the exported copy procedure can be used to copy bounded strings.

Is this a problem?

Probably not, the efficiency may differ, and Constraint_Error can be raised if the implicitly defined assignment is used, but it will never result in any nonsense values.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The first workaround is to use a private type even though the predefined comparison for equality and inequality return nonsense values. Instantiations are often made of generic packages with generic formal private types, even though some of the exported subprograms from the instantitated package which use the equality operation will return nonsense values.

The other workaround is to use a limited type even though the implicit assignment has to be replaced by a copy procedure.

This is a safe solution, but the code usually gets unnecessarily complicated and inefficient. You also lose the possibility to instantiate a lot of generic packages with generic formal private types, e.g. Sequential_Io and Direct_Io.

The limited type has to be converted to a non-limited type, e.g. by Unchecked_Conversion, before that is possible, and you again have a nonsense equality operator.

POSSIBLE SOLUTIONS:

Would the introduction of the possibility to declare an assignment operator for limited types solve this problem?

No, the declaration of an assignment operator would probably only be an aesthetic improvement of the language.

A copy procedure could just as well be used.
The copy procedure/assignment operator will not be visible inside a generic unit unless it is included as a generic formal subprogram.
(This discussion might of course be wrong, since I have not seen the revision request I am talking about.)

The suggested solution is therefore this:

Add a semilimited type to the standard which work as private type in respect to implicit assignment, and as a limited type in respect to the predefined comparison for equality and inequality.
It should be possible to have generic formal semilimited types.

(A simpler name than "semilimited" might be desirable.)

As a consequence of this change, should the generic formal type in the standard generic IO packages Sequential_Io and Direct_Io be made semilimited private instead of private.

```
generic
        type Element_Type is semilimited private;
    package Direct_Io is

        type File_Type is limited private;

        ...

        end Direct_Io;

        ...

        subtype Line is String_Utilities.Bounded_String
            (The_Size => 80);

        package Line_Stacks is new (Item => Line);

        subtype Line_Stack is Line_Stacks.Stack
                    (The_Size => 20;

        package Line_Stack_Io is new Direct_Io
                (Element_Type => Line_Stack);
```

Notice that the introduction of a semilimited private type is upward compatible with the current Ada standard. No Ada code will have to be rewritten due to this change.

It will probably not be a major problem to implement semilimited types, when private types and limited types already exists.

CAPABILITIES FOR DESCRIBING OBJECTS ARE DISTRIBUTED ACROSS TWO SEPARATE LANGUAGE CONCEPTS

DATE: October 11, 1989

NAME: William Thomas Wolfe

ADDRESS: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 7, 9, 12

PROBLEM:

The various capabilities for describing objects are distributed across two separate language concepts, packages and tasks.

IMPORTANCE:

CONSEQUENCES:

Objects are more difficult to describe directly, sometimes requiring a combination of tasks and (generic) packages.

Also, the fragmentation complicates Ada and makes it seem difficult to learn. Integrating the concept of a task and the concept of a package into the concept of an object would considerably simplify the language.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Integrate the concept of a package and the concept of a task into a single concept, "object". The rest of Ada would serve to assist with object implementation.

MAKING DERIVED SUBPROGRAMS UNAVAILABLE

DATE: October 22, 1989

NAME: Arnold Vance

ADDRESS: Afflatus Corp.
112 Hammond Rd.
Belmont, MA 02178

TELEPHONE: (617) 489-4773
E-mail: egg@montreux.ai.mit.edu

ANSI/MIL-STD-1815A REFERENCE: 7, 3.4

PROBLEM:

When defining a derived type in a package specification, it may not be desired to make all of the derived subprograms visible. There is no way to suppress the undesired derived subprograms.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

For each of the undesired subprograms, a dummy subprogram can be written which either does nothing or raises some exception. Doing nothing probably hides a software error. Raising a predefined exception (like program_error) is probably not a good solution; nor is defining an exception in each package especially for this situation. Perhaps the best solution currently is to define an exception in a library package so that all packages that need to raise it may do so.

POSSIBLE SOLUTIONS:

Allow a derived subprogram to be made unavailable. The suggested syntax is:

```
basic_declaration ::=  --see 3.1
        ...
        | subprogram_unavailable_declaration
subprogram_unavailable_declaration      ::=
        subprogram_specification is null;
```

The following is an example using such a construct:

```
package pkg1 is
        type t1 is private;--desired parent type

        procedure op1(p:t1);
        procedure op2(p:t1);
        procedure op3(p:t1);
        end pkg1;
with pkg1; use pkg1;
```

```
package pkg2 is
        type t2 is new t1; -- want to export op1 & op3 but not op2
        procedure op2(p: t2) is null; -- op2 not visible or derivable
        end pkg2;
```

The notion of derivability defined in 3.4(11) would need to be modified to account for subprograms made unavailable.

STANDARDIZED PACKAGE-LEVEL MUTUAL EXCLUSION

DATE: October 16, 1989

NAME: Samuel Mize

DISCLAIMER:

The views expressed are those of the author and do not necessarily represent those of Rockwell International.

ADDRESS: Rockwell/CDC
3200 E. Renner Road M/S 460-220
Richardson, TX 75081

TELEPHONE: (214)705-1941

ANSI/MIL-STD-1815A REFERENCE: 7, 9

PROBLEM:

Many activities need to be mutually exclusive, for example updating a global data structure or accessing a peripheral. The only facility Ada provides for this mutual exclusion is to encapsulate all such activities in a synchronization task. Such a task can be difficult to program correctly, especially if its operations affect more than one object requiring mutual exclusion (for example, operations moving data between several peripherals).

Writing explicit tasks to synchronize mutually-exclusive activities adds a number of extra tasks to the design. THese extra tasks can, in some cases, create significant task switching maintenance costs. If hidden in a package body, they can create timing or deadlock problems which will be hard for the package user to understand and debug (impossible, if the body is not available for inspection).

According to the Ada Rationale [1], "The monitor solves the exclusion problem but not the message problem." The Ada rendezvous solves the message problem, but requires manual coding of simple mutual exclusion.

IMPORTANCE: IMPORTANT

Buffers, peripherals which should be used by only one task at a time (e.g. Text_IO), and many other sorts of data-oriented transactions need to be clearly encapsulated without an artificial synchronization task. This should be visible in the package specification, so a user of the package can avoid code sequences that may create deadlock.

CURRENT WORKAROUNDS:

1. Write a task that encapsulates all the mutually-exclusive activities, and access them as entries.

2. As 1, but encapsulate the task in a package. This way, the task structure can be changed, if necessary, without affecting the user's interface to the resource.

3. Use pragma SHARED to implement mutual-exclusion semaphores.

All workarounds require manual coding of mutual exclusion. If exclusion must be maintained on more than one resource, this can lead to a rather Byzantine synchronization task, or to having a "semaphore" task for each resource, with a lock and unlock entry (this is a simple but very error-prone approach). Workarounds 1 and 2 add the overhead of an unnecessary check for task switching at each mutual-exclusion transaction. Workaround 2 provides a cleaner and more easily-maintained design (unless the hidden task creates a deadlock or timing problem, in which cases the maintenance programmer doesn't even know it is there).

POSSIBLE SOLUTIONS:

Provide a language-based way to define packages with mutual-exclusion constraints.

One approach would be to extend the pragma SHARED to composite objects. This is unattractive for a variety of reasons. For example, a conceptually indivisible transaction may require several operations on a given object.

The preferred approach is to extend the definition of subprograms in packages to include a specification of mutual exclusion.

This could be done without adding new reserved words by extending the meaning if the reserved word "XOR," usable only in a package specification. No subprogram with XOR in its specification (after the parameters, before the semicolon) would start to run while another XOR-exclusive procedure from the same package is in process (including if its task is suspended). In effect, there is a single (monitor-style) entry queue for all mutual-exclusion procedures, and they all behave like entries into one task. A lower-priority procedure running a mutual-exclusion subprogram would complete before a higher-priority procedure could start any other mutual- exclusion subprogram from that package.

To handle multiple resources needing exclusion in a single package, an optional parenthesized list of identifiers may follow the XOR. In this case, when a mutual-exclusion subprogram is running, mutual exclusion applies only to subprograms whose lists include any of the identifiers on the currently-running subprogram's list. In effect, there is a monitor-style entry queue for each identifier used in XOR lists.

This is made clearer by the following example:

```
-------------------------------------------------------
package Mutex_Example is

        type Item_Type is ...;
        type Trid_Type is ...;

        procedure Read_DiskA (I : out Item_Type) XOR (Disk_A);
        procedure Write_DiskA (I : out Item_Type) XOR (Disk_A);

        procedure Read_DiskB (I : out Item_Type) XOR (Disk_B);
        procedure Write_DiskB (I : out Item_Type) XOR (Disk_B);

        procedure New_Trid (T : in Trid_Type) XOR (Trid);

        procedure Transport_Trid_Data XOR (Disk_A, Disk_B, Trid);
```

```
end Mutex_Example;
-----------------------------------------------------------
```

The example shows a set of transactions involving three mutual-exclusion resources. For instance, a read from disk A may not begin while someone else, even of a lower priority, is writing to disk A, or is using it to transport a Trid. Similarly, Transport_Trid_Data waits to execute until it has exclusive control of all 3 resources. Consider the following call sequence:

task A calls Read_DiskA
task B calls Transport_Trid_Data
task C calls Read_DiskB

If task A has not been completed Read_DiskA, then task C cannot go ahead and do Read_DiskD, because it is in the Disk_B queue behind task B (task B has locked that resource).

Because of this potential problem, conditional and timed entry call syntax may be made usable with mutual-exclusion subprogram calls.

Of course, the problem is not even visible, let alone solvable, if the mutual exclusion is provided with one or more synchronization tasks inside Mutex_Example (as would be necessary now).

PRIORITY INVERSION:

If Ada9X task entry queues are changed form FIFO to priority-based to prevent priority inversion, the queues on mutual exclusion identifiers should also be priority-based. This requires a more complex resolution of mutual exclusion on several resources.

A lower-priority task will not be allocated resources that a high-priority task needs if the higher-priority task is also waiting on a different resource. Using the example package, consider the following call sequence:

task A (any priority) calls Read_DiskA and starts it
task X (any priority) calls Read_DiskB and starts it
task C (low priority) calls Read_DiskB
task B (high priority) calls Transport_Trid_Data
task X completes Read_DiskB

Task B entered the queue after task C, but will still keep C from running when task X completes. Task B's lock on the resource of a higher priority than task C's.

ABORTING TASKS:

When a task is executing a mutual exclusion subprogram and gets terminated, the completion state of the procedure is undefined. The next subprogram to try to use a procedure protected by any of the same mutual-exclusion identifiers will have an exception raised just after it enters. Note that the exception may be handled by an exception handler.

BENEFITS OF PROPOSAL: This language facility provides the following advantages over hand-coding synchronization tasks:

1. A possible source of programmer error (coding synchronization tasks) is eliminated.

2. Run-time efficiency may be gained by eliminating extra checks for task switching.

3. The package specification indicates mutual exclusion constraints, helping the user avoid deadlock. Some activities must create the potential for deadlocks; these must be correctly sequenced in the "customer"tasks to prevent deadlock.

   With a synchronization task, these constraints may be deduced from its specification. However, a task cannot be a library unit (must be in a package or procedure body to compile). So, users of a package encapsulating a system utility whether there are any hidden synchronization tasks, especially if the package body is proprietary and is not provided.

4. Mutual exclusion requirements for several interacting resources can be shown in one package specification. With current techniques, if transactions using several resources are encapsulated in a package to ensure their correct usage, its deadlock constraints are invisible to the user.

REFERENCES:

Ichbiah, Jean D.; Barnes, John G. P.; Firth, Robert J.; Woodger, Mike. Rationale for the Design of the Ada Programming Language. Honeywell Systems and Research Center, 1986. Section 13.3.1.

CHANGING PACKAGE TYPES INTO CLASSES WITH INHERITANCE

DATE: August 18, 1989

NAME: Goran Karlsson

ADDRESS: Bofors Electronics AB
Nettovagen 6
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 222 90

ANSI/MIL-STD-1815A REFERENCE: 7

PROBLEM:

Ada is not a true object oriented language.

IMPORTANCE: ESSENTIAL

To object oriented developers.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The package type concept has been introduced in an earlier revision request, however I do not at this moment remember who wrote that revision request.

The syntax for a package type was as follows:

```
package type package_name is

        package_body

end package_name;
```

True classes have inheritance. There are two distinct types single inheritance in which there may be only one parent to each derived class, and multiple inheritance in which there can be multiple parents to a class.

A possible syntax for such a package type using single inheritance would be:

```
package type package_name PARENT parent_package is

        package_body

end package_name;
```

For multiple inheritance one could use:

```
package type package_name PARENTS parent_package
                                                  -"-
                                                  parent_package is

        package_body

end package_name;
```

One should however keep in mind that multiple inheritance is more complicated to implement.

When using inheritance it should be possible to use dynamic binding between derived classes of a parent class, that is different derived classes from the same parent class can be used in place of each other.

Example:

```
PACKAGE type parent_class is
end parent_class;

PACKAGE type derived_class_1 PARENT parent_class is
end derived_class_1;

PACKAGE type derived_class_2 PARENT parent_class is
end derived_class_2;

variable1 : derived_class_1;
variable2 : derived_class_2;

variable1 := variable2; -- should be a legal statement
```

However a problem arises when both derived classes have procedures or functions with identical names and this procedure is called after such an assignment.

```
PACKAGE type parent_class is
end parent_class;

PACKAGE type derived_class_1 PARENT parent_class is

        PROCEDURE a;

end derived_class_1;

PACKAGE type derived_class_2 PARENT parent_class is

        PROCEDURE a; -- different from derived_class_1.a
end derived_class_2;

variable1 : derived_class_1;
variable2 : derived_class_2;
```

variable1 := variable2;

variable1.a; -- What procedure is called?

It is clearly necessary that this decision should be made at runtime. A possible implementation of this is to let the compiler add a table of pointers to all procedures and functions that have identical names. Then the call is made through this table, adding a single level of indirection.

In order to tell the compiler what procedures and functions should be put into this table one can add declarations in the parent class. In the two object oriented languages that I know of (SIMULA and C++) the reserved word VIRTUAL is used to indicate that the procedure or function called shall be determined at runtime. In this case:

```
PACKAGE type parent_class is

        VIRTUAL PROCEDURE a;

end parent_class;

PACKAGE type derived_class_1 PARENT parent_class is

        PROCEDURE a;

end derived_class_1;

PACKAGE type derived_class_2 PARENT parent_class is

        PROCEDURE a; -- different from derived_class_1.a

end derived_class_2;

variable1 : derived_class_1;
variable2 : derived_class_2;

variable1 := variable2;

variable1.a; -- derived_class_2.a is called
```

Final note: There must be a body even if a procedure or function is declared as VIRTUAL.

PACKAGE TYPES

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 7

PROBLEM:

There is no mechanism in Ada which produces an array of packages. It was recognized late in the development of Ada that there should be task types, largely so that tasks could be embedded inside of records and arrays, but no such facility was provided for packages.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

A task type can be declared instead of a package type, and then an array of objects of this type can be produced. There are several disadvantages:

- The task type cannot be a library unit, while a package type (presumably) could be.

POSSIBLE SOLUTIONS:

SUBCONTRACTING PROBLEMS

DATE: June 7, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 7.3

PROBLEM:

Subcontracting problems.

Is it essential to make subcontracting as hard as it is in Ada83? A caricatured picture of the situation can easily be given. Assume I specified:

```
package P is

        type T is limited private;
        E: exception;
        procedure P_ORDINARY (OBJ: T);
        procedure P_SPECIAL (OBJ: T);

private
        type T_OBJ:
        type T is access T_OBJ;
end P;
```

and assume that some other package already exists, and implements exactly what I need to implement P, except for the P_SPECIAL operation.

```
package P_EXISTING is

        type T is limited private;
        E: exception;
        procedure P_ORDINARY (OBJ: T);

private
        type T_OBJ:
        type T is access T_OBJ;
end P;
```

So, P_EXISTING is a very natural subcontractor.

One could imagine that the implementation of P in Ada9X would look like:

```
package body P is
        type T is new P_EXISTING.T;
        E: exception renames P_EXISTING.E;
        procedure P_SPECIAL (OBJ:T) is
        begin...end;
end P;
```

The Ada83 solution however would be:

```
with UNCHECKED_DEALLOCATION;
package body P is
        type T_OBJ is new P_EXISTING.T;
        procedure FREE is new UNCHECKED_DEALLOCATION (T_OBJ, T);
        procedure P (OBJ :T) is
        begin
                P_EXISTING.P (OBJ.all);
                FREE(OBJ);    If P_EXISTING.P deallocated its argument
        exception
                when P_EXISTING.E => raise E;
        end;

        procedure P_SPECIAL (OBJ:T) is
        begin...end;
end P;
```

Which does not look too catastrophic, except when one starts to use packages with more than 50 operations (a realistic package like TEXT_IO), then one starts to wonder whether one really needs to write (and read, debug,maintain) more than 400 lines of almost dummy code, that do nothing more than slowing down the execution, and increasing heap fragmentation and risk for human error (especially with UNCHECKED_DEALLOCATION).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

try to find an APSE that can do all this dummy work, please tell us if you found one....

POSSIBLE SOLUTIONS:

It seems that Ada specification are too much inspired on "I specify something hence I must do something", while software engineering asks for "I specify something hence I must ensure that something will be done, either by myself, or by any of my subcontractors". Hence, a natural solution would be to require that for each declarations in the package specification that needs to be implemented, preferably including incomplete type declarations, one and only one conform declaration needs to be made visible in the package body.

Note that all this might sound "Object Oriented", but in fact the approach suggests has nothing to do with OO aspects that are frequently considered unsafe (dynamic binding and polymorphism).

Another symptom of the lack of support for subcontracting is the private part of packages. If compilation

would be considered a verification step that also enables code generation as soon as possible but as a separate step, there wouldn't have been any real problem in delaying definition of private types and deferred constants to package bodies. The same distinction should be made between re-verification (on user demand) and re-code-generation (hidden).

In the example above this is also demonstrated: the body of P in Ada9X contains a full_type_declaration for T, and ideally this should replace the private part of P's specification (no one should care whether P_EXISTING.T is actually an access type to some other type T_OBJ, except for the code generator which should remain hidden).

TERMINATION CODE

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 7.3(2), 9.7.1(2)

PROBLEM:

The purpose of the elaboration phase in the life of a library unit is to reserve space for the declarative items declared by the library unit and, where appropriate, to assign them initial values. In addition, of course, the sequence of statements in a package body is executed during the elaboration of the package. [7.3(2)] This allows the programmer to cause initialization code to be executed prior to the actual execution of the main program. However, Ada provides no facilities at the other end--after the execution of the main program. In many cases, termination code can be used to "clean up" after the main program has finished its execution. It is the natural complement of initialization code.

This problem concerns packages and tasks. Subprograms can include termination code at the end of their sequence of statements. But packages and tasks are not reentrant; only one copy of the package or task persists until either it (in the case of tasks not dependent on the main program) or the main program (in the cases of packages and tasks dependent on the main program) terminates. In tasks termination code could be easily provided by permitting statements after a TERMINATE alternative. In packages the same thing could be accomplished by overloading the reserved word TERMINATE and allowing its inclusion after the BEGIN and before the EXCEPTION or the END of the package body.

Examples:

```
SELECT
  ACCEPT start;
OR
  TERMINATE;
  stack_manager.destroy(stack);
end SELECT ;

PACKAGE BODY file_manager is
BEGIN
  indexed_io.open(file, file_name, indexed_io.inout_file);
TERMINATE
  indexed_io.close(file);
end file_manager;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

For tasks, declare an entry that will terminate the task instead of relying on the terminate alternative. For packages, declare a procedure to be called at the end of the main program to provide the termination code.

POSSIBLE SOLUTIONS:

"OWN" VARIABLES IN PACKAGES

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 7.3 #3 and 7.3.2 #13 & 16

PROBLEM:

The OWN variable concept from Algol hardly belongs in a unified and integrated language design. It is inconsistent to have the concept only supported in packages, but not in other constructs. The concept hardly makes the language uniform. It is counter-intuitive when it appears like a declaration for a temporary with no warning syntax to make the items stand out from other candidates for elaboration. This is something that can cause major software maintenance problems when it is overlooked or is intended/interpreted by the programmer as a temporary, but it has other "global" performance characteristics.

First of all, such a capability is unnecessary--it can be an externally declared object. The elaboration rules need to be revised anyway for better control of data for static and controlled objects. The scoping rules for temporaries applied to a body should be the same throughout the language.

If this capability is what is desired, then it is better to have words like CONTROLLED or STATIC, like in PL-1, to indicate that the elaboration will not take place and the value will be retained. Then, you have a better use for CONTROLLED without introducing a new key word like OWN or a new concept in a language with too many special cases. The design should be such that the CONTROLLED (if that is chosen) can be used anyplace that a declaration can exist and not just in packages.

This OWN approach may not have meaning in generics. It also may cause problems with reentrance for **some unit importing/elaborating such a package.**

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: Avoid it in programming standards. However, a major problem can occur if an inexperienced Ada programmer misses the one sentence in the LRM overlooking the meaning of the one example in the LRM that explains this behavior.

POSSIBLE SOLUTIONS:

1. Delete the sentence and have the declared item take on the meaning of a temporary in a package.

2. Obtain the capability through correcting the elaboration problems where all declarations are re-initialized whether that is what is desired or not. (It is terribly inefficient and also leads to counter intuitive results).

PRIVATE SCALAR TYPES

DATE: July 21, 1989

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of the Ada UK.

ADDRESS: IPSYS plc
Marlborough Court
Pickford Street
Macclesfield
Cheshire SK11 6JD
United Kingdom

TELEPHONE: +44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 7.4

PROBLEM:

It is not possible to hide the declaration details of a scalar type definition. For example, the declarations:

```
type SIZE is range 1 .. 2048;
DEFAULT : constant SIZE := 256;
INCREMENT : constant SIZE := 16;
```

make visible the particular range of values for SIZE, and the precise values of DEFAULT and INCREMENT. This is not consistent with the role of packages as Abstract Data Types.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

There are two workarounds. The first is to declare the type explicitly as a scalar type and trust that users of the type will make use of the definition details through the use of 'FIRST, 'LAST, 'RANGE, etc., and that the constants will be used by their name.

Alternatively for a more 'abstract' approach a private type may be used, e.g.;

```
package PRIVATE_INTEGER is
   type SIZE is private;
   DEFAULT : constant SIZE;
   INCREMENT : constant SIZE;
   --operations as required
private

   type SIZE is range 1 .. 2048;
   DEFAULT : constant SIZE := 256;
   INCREMENT : constant SIZE := 16;
```

```
end PRIVATE_INTEGER;
```

Although this is a more secure formulation, the type cannot be used as a discrete range, e.g. in choices or index ranges.

POSSIBLE SOLUTIONS:

It is interesting to note that generic formal types offer a means of importing a richer class of abstract types than may be exported from a package. One possible solution to the problem, therefore, would be to extend the class of type definitions to match those of generic formal private types, which would allow the definition of private scalar types.

As an example:

```
package PRIVATE_SCALARS is
   type PRIVATE_DISCRETE is private (<>);
   type PRIVATE_INTEGER is private range <>;
   type PRIVATE_FLOAT is private digits <>;
   type PRIVATE_FIXED is private delta <>;
private
   --etc.
end PRIVATE_SCALARS;
```

The use of the keyword private could be optional to reinforce the similarity with generic type definitions.

[LIMITED] PRIVATE TYPES

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 7.4

PROBLEM:

LIMITED is so limited in capability that it is hardly worthy of adding complexity to the language design when all it does is prevent assignments and tests for equality/inequality. Therefore, the only remaining constructs that are available for limited private types are procedures for which those restraints are known and should automatically be prevented by the compilation system. Limited and Limited private just do not provide the desired information hiding capabilities that a modern language needs.

Also, a seldom used construct like limited presents maintenance and training problems even for experienced Ada programmers. They cannot remember the difference between Limited private and Private. There is nothing in the word "Limited" that suggests "how" or "what" is limited. The rules in 7.4.3 #5 .. 8, # 11 are too specialized to be useful. The programmer has to go read code (several packages away) to determine how to use the procedures defined under limited private types.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Rule it out in programming standards or provide massive amount of internal comments.

POSSIBLE SOLUTIONS:

Major work needs to be done on Private types to make it more useful than masking characteristics of printer peripherals for a programmer. It is not straightforward when the programmer has to re-invent the wheel by re-defining operations over the private types. Long units or modules of the same things, e.g., arithmetic definitions on each private type, are not in the direction of maintainability, productivity, and reliability.

LIMITED should probably be deleted as it doesn't support the object oriented programming that is desired. If packages and components within packages can be imported with WITH and assigned attributes such as IN, IN OUT, and OUT with package component selection, then Limited isn't needed. (See also, Mowday-014)

TOO MANY SPECIAL SEMANTICS SURROUNDING USE OF PRIVATE TYPES

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 7.4

PROBLEM:

There are several problems impacting software design when a programmer has to use private types for information hiding. First, the programmer gets to re-create operations allowed on private types (e.g., add, subtract). There's no shorthand way to give permission for standard or pre-defined operations to be allowed. Next, the rules are too specialized to be easily remembered. Combining private with generics is touchy. Also, discriminant and generics should not have so many special cases. Thirdly, from the syntax and semantics in this paragraph, the reader would never come to that conclusion given in 7.4.1 #6.

IMPORTANCE: IMPORTANT

The visible part needs to be very clear. The information hiding should truly be in the direction of code clarification capability and minimizing contention. In many ways the syntax for private types is the only mechanism for the true expression of the type constraints and user defined attributes that could be achieved by a more expressive declaration capability.

CURRENT WORKAROUNDS:

Avoid private types through programming standards as the impact to maintenance is too high. There are too many special cases for high quality programs. Provide the verbose axiomatic type declarations in private types for overloading the standard operations at a very high level. (Note that this adds source lines but no code.)

POSSIBLE SOLUTIONS:

Major rework to private types to provide the object oriented needs of today's program designs. Discriminants, parameterization, and generics need to be reworked to make them more uniform with other constructs throughout the language.

SEPARATELY COMPILABLE DATA OBJECTS

DATE: September 26, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 7.4

PROBLEM:

Many systems, but particularly real-time systems, need adaptation or configuration data which is supplied at program link or load time. In general, this data must be visible to many library units. In order to minimize the time between changing the adaptation data and re-execution of the program, it must possible to alter it without requiring recompilation of any part of the system except the module containing the declaration. Often this data should be constant once it is elaborated, so it should be possible to pre-elaborate it. It is desired that the solution be relatively portable.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Data initialization may be supplied using another language, based on fixed addresses for the data, and requiring complete information about the representation and layout of that data. This approach is rather fragile and certainly not portable.

POSSIBLE SOLUTIONS:

Allow deferred initialization of objects. In particular, allow deferred constant declarations for non-private types and deferred variable declarations, whose value can be fully defined either in the private part of the package or, more importantly, in the declarative part of the package body (rather than the sequence of statements of the package body). For example:

```
package P is

    type T is range 1 .. 10;
    O1 : constant T; --deferred initialization of constant object
```

or, in order to distinguish initialization in the body from initialization in the private part, possibly:

```
    O1 : constant T := <>; -- deferred initialization of constant object
```

```
        O2 : T := <>; -- deferred initialization of variable object
end P;

package body P is

        -- The following initializations could be pre-elaborated:
        O1 : constant T := Value;
        O2 : T := Value;
end P;
```

It would, of course, be possible to assign a value to O2 using a function call, either at the point of its declaration or in the sequence of statements in the package body, but this would not be possible to pre-elaborate.

This proposal is similar to, but more comprehensive, than Ada9x Revision Request number 0093. It is intended to have the effect of providing separately compilable data objects.

PRIVATE TYPES ARE TOO PRIVATE

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.,
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 7.4

PROBLEM:

It is a recurring problem in the formulation of interface specifications that some types are pervasive throughout the interfaces offered. These types may have to be (limited) private to the user of the interfaces. Yet, it can be exceedingly inconvenient to combine all interfaces whose implementation requires knowledge of the internals of these types in a single package to establish the necessary visibility into the internals of the private type. Rather, a grouping by functional area or capabilities is desirable. However, such capability packages cannot be implemented cleanly, due to the obvious visibility problems. Furthermore, many of the employed implementation schemes falter when these private types are needed as actual parameters to generic units required for the implementation of the types themselves and visible to the users of the interfaces.

Examples of large interfaces that had to contend with these problems are MIL-STD-1838(a), PCTE Ada Binding, X-Window binding and others.

Current work-arounds range from unchecked_conversion techniques to hidden type implementation packages, assumed to be known only to the implementors of the interfaces but not the user of the interfaces. The former is obviously fraught with danger; the latter quickly becomes unwieldy and requires numerous type conversions. Neither addresses the mentioned issue of generic instantiations with such private types cleanly.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

No safe/portable write-arounds.

POSSIBLE SOLUTIONS:

There should be a way for "friendly packages" to gain access to the information contained in the private part of a package.

A possible and rather simple solution would be to extend context clauses of packages such that "friendly access" to the private part of an imported unit can be established, e.g.,

```
with foo;                    -- normal non-private visibility
with private bar;            -- grants visibility to private part of bar -- as well
package friend is ....
```

COMPLETION OF TYPES BY SUBTYPES

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 7.4

PROBLEM:

It would be convenient if incomplete and private types could be completed by a subtype declaration rather than a full (derived) type declaration.

Completion by subtypes would avoid the necessity of frequent type conversions between derived types, where such type equivalence is appropriate.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Type conversions.

POSSIBLE SOLUTIONS:

PRIVATE PART SHOULD BE OPTIONAL

DATE: September 13, 1989

NAME: R. David Pogge

ADDRESS: Naval Weapons Center
EWTES - Code 6441
China Lake,CA 93555

TELEPHONE: (619) 939-3571
Autovon 437-3571
E-mail: POGGE@NWC.ARPA

ANSI/MIL-STD-1815A REFERENCE: 7.4.1

PROBLEM:

If an Ada package specification uses a private or limited private type, the full definition of that type must be given in the private part of the package specification. This is undesirable because it forces implementation details to be given in the package specification, defeating the purpose of the separation of the package specification and package body. If the representation of the private type is changed, the package specification must be recompiled. This forces all dependent units to be recompiled.
For a more complete discussion of the problem, see "On Inclusion of the Private Part in Ada Package Specifications" by Sitaraman Muralidharan, Ohio State University. This paper is published on page 188 of the 7th Annual National Conference on Ada Technology proceedings. Muralidharan argues that the private part is undesirable, and unnecessary for a compiler to generate efficient code.

IMPORTANCE: IMPORTANT

This is IMPORTANT. It is not ESSENTIAL because software development can easily continue using the normal workaround. It would, however, make the language more consistent with the design intent and improve productivity by eliminating unnecessary re-compilations.

CURRENT WORKAROUNDS:

Recompile all dependent units whenever a change is made to the private part of a package specification.

POSSIBLE SOLUTIONS:

Change Section 7.4.1 paragraph 1 to read

> If a private type declaration is given in the visible part of a package, then a corresponding declaration of a type with the same identifier must appear as a declarative item EITHER of the private part of the package SPECIFICATION OR OF THE PACKAGE BODY. The corresponding
> . . .

The private part must be maintained for compatibility with existing programs, but programmers should be encouraged to put the full type declarations in the package body rather than the private part. Perhaps the

private part can be eliminated in 1815C.

FULL DECLARATION OF PRIVATE TYPES

DATE: September 29, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd,
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 7.4.1

PROBLEM:

The full declaration of a private type should give maximum freedom to the implementer of the type to implement the type in a flexible and efficient manner consistent with the abstraction provided by the private mechanism. Currently this is not so.

As a trivial example considers the problem of implementing a varying string. Two obvious approaches present themselves: to use a discriminated record and thus impose some maximum length, or to use an access type and thus incur the overhead of a pointer. Thus consider.

```
subtype STRING_SIZE is INTEGER range 0 .. MAX;

type V_STRING(N: STRING_SIZE:=0) is
    record
        S: STRING(1 .. N);
    end record;

type A_STRING is access STRING;
```

As a more elaborate example, what we would really like to do is to implement a string type as a private type in such a way that a very short string is held directly and longer strings are held as an access value. Clearly we need to distinguish between the two implementations and the obvious technique would be to use a discriminated type as the full type declaration with discriminant selecting between the two implementations. Thus something like:

```
type SUPERSTRING is private;
...
type KIND is (ZERO, ONE, TWO, THREE, MORE);

type SUPERSTRING(SIZE: KIND:= ZERO) is
    record
                    CASE SIZE is
        when ZERO =>
            null;
        when ONE =>
```

```
        S1: STRING(1..1);
    when TWO =>
        S2: STRING(1..2);
    when THREE =>
        S3: STRING(1..3);
    when MORE =>
        A: A_STRING;
    end case:
end record;
```

This rather blunt formulation coupled with an appropriate record representation clause should enable a short string of up to three items to be held in one word with a fourth byte holding the discriminant.

However, this is not allowed since the full type declaration of a private without discriminants may not be a type with discriminations (even if there is a default value of the discriminant). There is no way to implement what we want - we are always forced to use an access type and incur the basic overhead.

An analogous restriction used to apply to the use of a discriminated type as an actual type corresponding to a private formal type of a generic unit. This was forbidden even if the actual type had a default value for the discriminant. This restriction was relaxed by AI-00037. The consistency between the two situations has now been broken.

The subject needs revising - see the discussion reference in AI-00037.

IMPORTANCE: IMPORTANT

This is really quite important. The abstraction is currently flawed.

CURRENT WORKAROUNDS:

One always has to use access types and this incurs an additional overhead.

POSSIBLE SOLUTIONS:

Allow a discriminated type with defaults to be a full type corresponding to private type without discriminants.

CONTROLLING EXISTENCE OF LIMITED PRIVATE TYPE OBJECTS

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 7.4.1

PROBLEM:

Controlling Existence of limited private type objects.

A lengthy proposal as this one clearly needs a short abstract to motivate the reader. The problem observed has to do with efficient (space & time) and safe memory recuperation, which is currently not available in most compilers. Many real-time system designers simply don't care, since they often don't depend on it. Large information processing applications do, e.g. CAD, CASE and scientific applications. The observation is that reference counting memory management is often a solution, but very hard to implement in a safe way. Implementing this is feasible (works very efficiently) but the solution is extremely hard to use. Important issues: this solution is a lot better than trashing or leaving recuperation control to the caller (like unchecked deallocation does), the solution reveals some design flaws concerning limited private types, the solution is useful for most (but not all) data structures, with a little help from the compiler the solution would become programmer friendly, the problem is a complex one and cannot be explained on one page.

The design flaw concerning limited private types pops up when limited private types are not used because assignment is not a sensible operation for the type (like a task type), but because memory management should be done under control of the implementation, i.e. when assignment is more than just moving bytes.

For this observation, I would like to refer to "Studies in Ada Style", Hibbart, Hisgen, Rosenberg, Shaw and Sherman, Springer Verlag, 1981. At page 39, they recommend that if equality and/or assignment should not be implemented by the predefined operators, one should use limited types (rather than using subprograms (ASSIGN_PROPERLY and COMPARE, and leaving the comparison and assignment can only call programmer-defined subprograms ASSIGN and"="). So, they recommend limited types not only when these operations should not be supported, but also when they should not be implemented by the implicit declarations. Other books I checked don't treat the problem formally (nor "Rationale for the Design of the Ada Programming Language" Ichbiah, Barnes, Firth, Woodger, nor "Software Engineering with ADA", Booch, nor "System Design with Ada", Buhr, nor "Ada for Experienced Programmers", Habermann and Perry). So, the design flaw could already be identified here: the software engineering requirement of independence between specification and implementation is violated. When the comparison and assignment operations should be available with their normal semantics, implementation considerations still have an overwhelming impact on the specification, and they really should not.

A practical problem with garbage collection implementation is that assignment can be easily defined by a procedure ASSIGN (not as nice as procedure ":=", but feasible), but destruction can only be implemented

by providing the programmer/user with a procedure DESTROY that he should call each time an object of the limited type will be destroyed. Objects of a limited type can get destroyed if the program leaves the scope of a declarative region that contains an object of the limited type or an object that contains a subcomponent of the limited type. Likewise dynamic objects can get destroyed by calling UNCHECKED_DEALLOCATION instantiations. So, the user responsible for calling the DESTROY operation for each reference he destroys. This includes indirect destruction of references that are destroyed by the compiler because their scope is left, or because an instance of UNCHECKED_ DEALLOCATION is called to destroy an object that contains a (sub)component of the limited type. One of the most irritating problems is that programmers often forget to call this destruction operation, and it is very hard to protect against this error, the result is undetected trashing.

Now let's look at an example. E.g. a private type POINT, dynamically allocated just to demonstrate the problem:

```
package POINT is
        type T is private;
        type T_LIST is array (NATURAL range <>) of T;

        function MIRRORED (A_POINT, ORIGIN:t) return T;
        function CARTESIAN (X,Y: REAL) return T;
        function "+" (LEFT,RIGHT:T) return T;
        function "*" (LEFT,RIGHT:T) return T;

        function SUM (POINT_LIST: T_LIST) return T;
        private
        type T_OBJ;
        type T is access T_OBJ;
        end POINT;
```

one might write (meaningless but typical code):

```
        declare
        OLD_ORIGIN: constant POINT.T := MIRRORED (SOME_POINT, CARTESIAN (10.0,20.0));
        begin
        NEW_ORIGIN:= OLD_ORIGIN + (POINT1*OLD ORIGIN);
        OLD_ORIGIN:= SUM ((NEW_ORIGIN, OLD_ORIGIN, POINT1));
        end;
```

This block statement above will trash 5 objects of type T_OBJ for each execution. This solution looks very nice, but is purely academical in view of the unsolved memory management problems. As long as users may freely copy points using assignment statements, memory management will be an impossible job to do.

Well, currently there is no way you can allow the user to use the code show above when changing to automatically controlled memory management and hence a limited private type. The basic problem already appeared in the problem presentation above, where the referenced document, recommended that if equality and/or assignment should not be implemented by the predefined operators, one should use limited types (rather than using subprograms ASSIGN_PROPERLY and COMPARE). This recommendation is very sensible in the current situation, but on the other hand, it is a clear violation of modularity and information hiding: an implementation detail crept up into the specification, severely affecting the users! For instance, if after some usage of a non-limited private type, one decides that assignment still needs to be supported (specification doesn't change) but that one can no longer use the predefined assignment as implementation (implementation changes), one needs to change to a limited private type, and support a procedure ASSIGN (LEFT: in out T; RIGHT: T).

This minor implementation change requires all users to:

* modify all their L:=R; statements in ASSIGN (L,R),

* replace all constant declarations by variables and initializing ASSIGN calls (possible requiring addition of a package body), similar for all object declarations with initial values.

* replace all aggregates by variables that are initialized component by component

* replace all generic instantiations that associate the type with a formal generic non-limited private types with an instantiation of an equivalent generic unit that uses limited private types and additional formal subprograms like ASSIGN and DESTROY, and hence to implement such generic units if they are not yet available.

* to modify all mutant type declarations that contain the (now limited)type as a subcomponent. Indeed, mutants can only mutate by complete (predefined) assignment which is no longer available.

* where the type was used as a component in a record type, and where that component was given an initial value in the record declaration, remove this initial value. Add a initialization subprogram that assigns this initial value to the record type, and require a call to this subprogram as first use of each declaration of this type.

* to modify all mode out parameters of the type to mode in out.

* to modify all generic formal parameters of mode in to mode in out.

* ...who knows exactly all effects of this kind? Just try to enumerate these effects yourself.

* and lastly, all former non-limited types that contain the limited type as a subcomponent become limited too, and since they were former non-limited types, assignment should stay available for them. Hence, all users of such types need to undergo the same transformation.

I would like to continue using memory management as an example. In the example shown above, and in all types that are not involved in cyclic references, reference counting is a simple and efficient solution for memory management. One might hope that this situation is a unique, exotic example which will disappear as soon as compilers provide good memory management facilities.

This is however not the case, as this paragraph will point out (skip it if you are already convinced). A slightly more complex example is a symbol table, a da[illegible]ucture that is used to efficiently manage identifiers and/or long names: storing them in a virtual me[illegible]or in secondary storage, comparing them, and possibly associate information with them. The additiona[illegible]nplexity one has to take into account here that the identifier objects are both referenced from data structures managed by client packages and by data structures that are needed to efficiently retrieve an identifier or name given its ascii representation. Clearly symbol table objects should be deallocated when there are no more client package references, and before the deallocation, all internal references to is should be removed. Essential is here that the "event" that triggers the action is not the reference count dropping to zero (which could be known to an (1983) Ada compiler with Garbage collection), but the reference count dropping to the number of internal references (typically a count of 1 for hash tables or binary tree). This to conclude that Ada would be a lot more powerful if reference counts could be easily and safely controllable by the programmer, which implies that he gets informed about each copy and each destruction of a value of the type he controls.

Generic units for programmer controlled formal types are harder to write than one would expect at first sight, because the programmer is also responsible to patch up the result of compiler made copies. For instance, mode in parameters are copies from the actuals made by the predefined assignment, so when we are using programmer controlled assignment, we still need to add an operation like COMPLETE_ASSIGNMENT_OPERATION (IN_PARAMETER), and also a destroy operation that does not set its argument to null, such that it can be used on mode in parameters. and suck operations need to be supported by the package defining the limited private type (another modification to the specification), and passed to the formal subprograms of the new generics. It should be clear that considering this amount of complexity, a lot of programmers will either forget about memory management (often equivalent to guaranteeing STORAGE_ERROR) or shift the responsibility to the application programmer (almost guaranteeing return to C-style programming, spending lots of time in search for the dangling reference access that initiated an almost invisible chain of memory corruptions).

Let's go back to the example and see what happens if we choose to change as little as possible to the user's code keeping Ada as is. Although more sophisticated solutions are possible (which improve things a little but do not alter the basic reasoning here), let's assume that all functions destroy their arguments, and that a user needs to make copies where he doesn't want this destruction. Copies are made by function calls that return their argument after reference count increment.

```
package POINT is
        type T is private;
        type T_LIST is array (NATURAL range <>) of T;

        function COPY (A_POINT:T) return T;
        --returns a copy of A_POINT;

        procedure ASSIGN (TO_POINT : in out T; VALUE:T);
        --Destroys VALUE

        procedure DESTROY (A_POINT : in out T);

        function MIRRORED (A_POINT, ORIGIN:t) return T;
        function CARTESIAN (X,Y: REAL) return T;
        function "+" (LEFT,RIGHT:T) return T;
        function "*" (LEFT,RIGHT:T) return T;
        --All these functions implicitly destroy their arguments of type T.

        function SUM (POINT_LIST: T_LIST) return T;
        private
        ...
        end POINT;
```

The user should then write something like:

```
        declare
        OLD_ORIGIN :POINT.T;
        begin
        ASSIGN (OLD_ORIGIN, MIRRORED (COPY(SOME_POINT), CARTESIAN (10.0,20.0)));
        ASSIGN (NEW_ORIGIN, COPY(OLD_ORIGIN)+(COPY(POINT1)*
        COPY (OLD_ORIGIN)));
```

```
        declare
        SUM_ARGUMENTS :POINT.T_LIST (1..3);
        begin
        ASSIGN (SUM_ARGUMENTS(1), NEW_ORIGIN);
        ASSIGN (SUM_ARGUMENTS(2), OLD_ORIGIN);
        ASSIGN (SUM_ARGUMENTS(3), POINT1);
        ASSIGN (OLD_ORIGIN, SUM_ARGUMENTS));
        end;

        DESTROY (OLD_ORIGIN);
        end;
```

Several disadvantages of this package are obvious:

* Assignment statements are less visible and one extra pair of parentheses is shown.

* One loses the "constant"as a write protection on OLD_ORIGIN.

* The SUM function can no longer be called using an aggregate. Using a proper programming style, the one line call to SUM explodes into a eight line block statement.

* Most serious is the fact that forgetting the COPY or forgetting the DESTROY will not result in compilation or runtime errors! Forgetting the copy results in a dangling reference access (functions cannot set their arguments to null, so using the argument results in undefined execution that might work a lot of times but not always), forgetting the destroy results in garbage creation.

* When one compares the user's code with the former use of a nonlimited type one it utterly tempted to go back to non_limited, types, delaying the memory management problems until later times. An attitude that clearly should not be encouraged the way it is encouraged now.

A second choice implies complete decompilation of expressions to procedure calls, and again, memory management completely dependent on the discipline of the application programmer:

```
package POINT is
        type T is private;
        type T_LIST is array (NATURAL range <>) of T;

        procedure ASSIGN (TO_POINT : in out T; VALUE:T);
        procedure DESTROY (A_POINT : in out T);

        function MIRRORED (A_POINT, ORIGIN:T) return T;
        function CARTESIAN (X,Y: REAL) return T;
        function ADD (LEFT,RIGHT:T; RESULT: out T);
        function MUL (LEFT,RIGHT:T; RESULT: out T);

        function SUM (POINT_LIST: T_LIST; RESULT: out T);
        private
        ...
        end POINT;
```

in which case the program example becomes:

```
declare
OLD_ORIGIN :POINT.T;
begin
declare
TEMP:POINT;
begin
CARTESIAN (10.0,20.0, TEMP);
MIRRORED (SOME_POINT, TEMP, OLD_ORIGIN);
DESTROY (TEMP);
end;

declare
TEMP:POINT;
begin
MUL (POINT1,OLD_ORIGIN, TEMP);
PLUS (OLD_ORIGIN, TEMP, NEW_ORIGIN);
DESTROY (TEMP);
end;

declare
SUM_ARGUMENTS :POINT.T_LIST (1..3);
begin
ASSIGN (SUM_ARGUMENTS(1), NEW_ORIGIN);
ASSIGN (SUM_ARGUMENTS(2), OLD_ORIGIN);
ASSIGN (SUM_ARGUMENTS(3), POINT1);
ASSIGN (SUM_ARGUMENTS, OLD_ORIGIN);
end;

DESTROY (OLD_ORIGIN);
end;
```

Which will be considered by most programmers as more difficult to read, maintain and write. An extra disadvantage is the decompilation process that introduces the need for temporary variables that previously did not appear, all other disadvantages of the former approach remain valid here.

Conclusion: this is a real pain you-know-where

IMPORTANCE: IMPORTANT

Any major influence of implementation changes on specification change and hence change of client packages is also a mayor component of software production and maintenance costs.

CURRENT WORKAROUNDS:

No real workaround possible, one has to rely on programmer discipline (an approach with well known consequences). A very intelligent APSE could also do the job, but this would be the "preprocessor" solution.

POSSIBLE SOLUTIONS:

One might consider to extend the language to support all the private type operations for the limited private types, if the user is willing to present the compiler an implementation for these operations (assignment and destruction).

To forget for a moment how these implementations need to be recognized, let's assume that we pass this kind of information to the compiler by a pragma, a possible better solution will be suggested afterwards.

```
package POINT is
type T is limited private;

procedure ASSIGN (LEFT : in out T; RIGHT T;)
procedure DESTROY (OBJ: in out T);
pragma ASSIGNMENT_SUPPORT(ASSIGN, DESTROY);
function MIRRORED (A_POINT, ORIGIN:T) return T
function CARTESIAN (X,Y:REAL) return T;
private
type T_OBJ;
type T is access T_OBJ;
end POINT;
```

which would allow a user to write whatever he could write when the type was not limited:

```
declare
OLD_ORIGIN: constant POINT.T:- MIRRORED (SOME_POINT, CARTESIAN (10.0,20.0));
begin
NEW_ORIGIN:=OLD_ORIGIN + (POINT1*OLD_ORIGIN);
OLD_ORIGIN:=SUM((NEW_ORIGIN, OLD_ORIGIN, POINT1));
end;
```

A compiler could support this code by calling procedures ASSIGN and DESTROY each time he needs to make a copy of something (e.g. to implement an assignment statement or for its own copy operations, e.g. copy actual in parameters on the stack, copy values to an aggregate, etc...), and each time he destroys a value of the type (e.g. when popping the stack upon subprogram return or upon raising an exception, when UNCHECKED_DEALLOCATION is called for an object that contains a value of the type, when a mutant mutates to a variant that no longer has a subcomponent of the type, etc...).

An additional (and maybe less obvious) advantage of this approach com be found in the package body, implementing the limited type. If the package body uses programmer controlled assignment (for instance "copy & increment reference count" for reference counting memory management), then it is of crucial importance that even in the implementation of the package all ":=" operations refer to the programmer controlled assignment, except within the implementation of the programmer controlled assignment operation itself.

This is very similar to the situation of programmer controlled comparison, where Ada supports exactly this approach : when"=" is declared explicitly, all references to the "=" operator are resolved to the explicitly declared "=", and the predefined"=" can only be accessed by using special tricks. Why not having the same safety for assignment, making use of the proper implementation easy and use of the improper implementation tricky.

Now, one still needs the predefined operator ":=" to be able to implement the programmer controlled assignment. A tricky way of accessing predefined assignment where the complete type declaration is visible

(in the package body) uses a generic unit that makes predefined operators visible:

```
generic
type T is private;
--The actual type, i.e. the full type implementing the limited type must no be limited.
package G_PREDEFINED is
function COMPARE (LEFT,RIGHT:T) return BOOLEAN;
procedure ASSIGN (LEFT: in out T; RIGHT :T );
pragma INLINE (COMPARE,ASSIGN);
end G_PREDEFINED;

package body G_PREDEFINED is
function COMPARE (LEFT,RIGHT:T) return BOOLEAN;
begin return LEFT+RIGHT; end;
procedure ASSIGN (LEFT: in out T; RIGHT :T ) is
begin LEFT:=RIGHT, end
end G_PREDEFINED;
```

The trick is based on:

> ARM 12.03(15) : within the instance a predefined operator or basic operation of a formal type refers to the corresponding predefined operation of the actual type associated with the formal type.

So this solution would permit the compiler to implement all assignment statements using the explicitly declared ":=", except within the generic units, where the assignment would follow rule 12.03(15).

If this would be considered still to much overhead, or if one would change 12.03(15), it seems natural to suggest a solution that covers both problems at the same time: definition of two more attributes:

TYPE'PREDEFINED_ASSIGNMENT(LEFT: **in out** T; RIGHT :**in** T); and TYPE'PREDEFINED_COMPARISON(LEFT,RIGHT:**in** T); which are available as type attributes for all non-private types (and hence only available within the package body where availability is determined by the full type declaration).

At last one might want something cleaner than a pragma to connect the assignment operation, possibly one should be able to directly declare procedure ":=", giving the application programmer the clear message that assignment is available but not implicitly declared. This would also be a more consistent solution since similarity between the assignment and comparison operator is stressed.

For the procedure "DESTROY" however a pragma is just fine, since in fact we don't want the application programmer to see the fact that a compiler calls destroy, this is purely an implementation issue.

In this situation the example becomes:

```
package POINT is
type T is limited private;

procedure ":=" (LEFT : in out T; RIGHT T;)
function MIRRORED (A_POINT, ORIGIN:T) return T
function CARTESIAN (X,Y:REAL) return T;
function "+"(LEFT,RIGHT:T) return T;
function "*"(LEFT,RIGHT:T) return T;
```

```
private
type T_OBJ;
type T is access T_OBJ;
procedure DESTROY (OBJ: in out T);
pragma CONTROLLED_DESTRUCTION(T,DESTROY);
end POINT;
```

One might imagine the procedure syntax would become:

```
procedure designator [formal_part]
```

where the only allowed operator_symbol would be ":=".

the assignment statement would be then be translated to ":="(LEFT,RIGHT) after the compiler has taken care of "special" assignment features like array assignment.

Other restriction would be similar to the restrictions imposed on operator overloading : procedure ":=" needs to have one in out parameter "LEFT" and one in parameter "RIGHT" both of the same limited private type.

Still more restrictions could prevent misuse of the feature, although these restrictions are of limited use:

procedure ":=" can only be defined if the suggested pragma CONTROLLED_DESTRUCTION is used, so one avoids that people think that the compiler would actually destroy the things he copied using ":=" without support from the implementor.

If the procedure ":+" and the destruction are both defined, then all objects of the limited private type should obey that for any variable A of the limited private type, a block statement such as:

```
declare
TEMP:LIM_TYPE;
begin
TEMP:=A; --Make a copy of A
DESTROY(A); --Get rid (recover memory) of the original
A:=TEMP; --Copy the value back to the original
DESTROY(TEMP --Get rid(recover memory) of the temporary copy
end;
```

might be inserted anywhere A is visible without influencing the result of execution, otherwise the program is erroneous. This should give the user a reasonable guarantee that the semantics of ":=" and DESTROY are "sensible".

Then, if these modifications would be included in the language, one still needs to adapt the language to this new situation: if the assignment and comparison are made explicitly available to limited types, then why still considering this type limited?

So, why not change:

> ARM 06.07(04): The explicit declaration of a function that overloads the equality operator "=", other than by a renaming declaration, is only allowed if both parameters are of the same limited type.

to:

> ARM 06.07(04): The explicit declaration of a function that overloads the equality operator "=",

other than by a renaming declaration, is only allowed if both parameters are of the same private type.

And add a similar rule for the assignment operator ":=".

In this case the language would not lose constants and expressions for such private types, since they would be no longer limited and a compiler could use the operation ":=" to initialize a constant object, copy a returned result on the stack, create an aggregate, to implement predefined assignment for composite types that contain the type as subcomponent. Similarly a compiler would use the operation DESTROY to delete a value.

And none of the far reaching modifications enumerated above will impact a project when one changes to programmer controlled assignment.

Upward compatibility is no issue since programs that do not use any of the suggested extensions cannot be affected by their presence.

The counter-argument that one should not over-complicate the language is also questionable. The first purpose of the language should support avoiding the trickiness of software development in real-life situations, which is demonstrated to be not fully achieved (unless someone has a solution to the problem described).

As a last remark, a lot of people are known to fear computers that generate assembly-level code that is not a trivial translation of the source code, and that such people would be afraid to see a compiler implementing predefined assignment of on type calling programmer-defined assignment of another. First, language features like exceptions, genericity, recursion and parallelism which are "accepted" now must have produced the same attitude in the past. I guess that well established software engineering principles should outweigh human emotion. Second. if people really do not want to use procedure ":=" and/or the suggested pragma CONTROLLED_DESTRUCTION, it is a trivial job for an APSE to verify that these features are not used. The reverse, making the APSE responsible for introducing these features in the language is not trivial an implies a major preprocessing step, with the well known drawbacks. Lastly, the suggested change might be insecure, but still a lot safer than the formerly accepted UNCHECKED_DEALLOCATION.

COMPLETE SEPARATION OF THE SPECIFICATION
AND IMPLEMENTATION OF A COMPONENT

DATE: June 23, 1989

NAME: Jeffrey R. Carter

ADDRESS: Martin Marietta Astronautics Group
MS L0330
P.O. Box 179
Denver, CO 80201

TELEPHONE: (303) 971-4850
(303) 971-6817

ANSI/MIL-STD-1815A REFERENCE: 7.4.1(1)

PROBLEM:

Software engineering principles require a complete separation of the specification and implementation of a component. Ada reflects this by not requiring recompilation of something using a component when only the implementation of the component has been modified. The private part of an Ada package specification is implementation information which appears in the specification and violates this principle.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Perform unnecessary recompilations.

POSSIBLE SOLUTIONS:

Eliminate the private part of Ada package specifications. The full type declaration of a private type should appear in the package body.

One way to achieve this is to have the declaration of an object of a private type provide storage for an address. All references to the object will automatically dereference the address. Code to allocate space for the object, assign the address, and perform any initialization must be added from the body at link time.

RESTRICTIONS TO USE OF PRIVATE TYPES

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 7.4.1(4)

PROBLEM:

Restrictions to use of private types.

ARM 07.04.01(04): Within the specification of a package that declares a private type and before the end of the corresponding full type declaration, a restriction applies to the use of a name that denotes the private type or a subtype of the private type that has a subcomponent of the private type.

One might argue that this rule is too much of a concession from the programmer to the compiler.

Hence, the question is "is the only purpose of rule 07.04.01(04) to prevent compilers to worry about a second pass (where they are typically doing at least 5 passes)?"

A few examples might illustrate that this rule is far from "intuitive" for the programmer, and makes his code harder to understand. One could imagine that a programmer writes:

```
with LINKED_LIST;
package SOMETHING is
        type T is private;
        package LIST is new LINKED_LIST (T); --not allowed

private
        type T is...;
end SOMETHING;

with SOMETHING;
procedure TEST is
        A_SOMETHING: SOMETHING.T;
        A_LIST:SOMETHING.LIST.T;
begin
        SOMETHING.LIST.CREATE(A_LIST);
        SOMETHING.LIST.INSERT(A_LIST, SOMETHING.CREATE(...));
end TEST;
```

Which does not compile.

Note that in this context, rule 07.04.01(04) implicitly destroys the possibility for nesting of packages without the need for needless repetition. The repetition is the result of copying the generic package specification of LINKED_LIST in the package SOMETHING and implementing each of these operations by a one-line subprogram, this means that each operation identifier in linked_list will appear once in the specification of SOMETHING and twice in its body).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Separate packages need to be used:

```
package SOMETHING is
        type T is private;

private
        type T is...;
end SOMETHING;

with LINKED_LIST;
package SOMETHING_LIST is new LINKED_LIST (T);

with SOMETHING;
with SOMETHING_LIST;
procedure TEST is
        A_SOMETHING: SOMETHING.T;
        A_LIST:SOMETHING.LIST.T;
begin
        SOMETHING.LIST.CREATE(A_LIST);
        SOMETHING.LIST.INSERT(A_LIST, SOMETHING.CREATE(...));
        end TEST;
```

POSSIBLE SOLUTIONS:

One might solve this problem by allowing a private part as a block within the package specification rather than as a list of basic_declarative items that can be placed only at the end of a package specification.

```
package_specification ::=
        package identifier is
                {basic_declarative_item|private_part}
end [package_simple_name];
private_part::=
        private
                {basic_declarative_item}
end private;
```

Although this would allow an easy way of avoiding any problem imposed by rule 07.04.01(04), this syntax would not be compatible with the syntax currently used (because of the absence of the "end private", combining both rules would result in a new syntax:

```
package_specification ::=
        package identifier is
                {basic_declarative_item|private_part}
        [private
                {basic_declarative_item}]
end [package_simple_name];
private_part::=
        private
                {basic_declarative_item}
        end private;
```

This syntax is not ambiguous (since private is required in "end private"), but of course, not easy to parse. Hence, the practical mind will probably prefer the incompatibility that can easily be solved by a use-once precompiler that adds a terminating end private.

Another alternative of course is to discard 07.04.01(04) entirely. In fact, this would affect compilers probably to a less extent than one would imagine at first sight. At first sight 07.04.01(04), together with the visibility rules, and type rules 03.03.01 and 03.08.01 avoids the possibility of directly recursive types (e.g. records containing themselves, which are only implementable if the recursion goes through a variant, and imply too much complexity for the compilers). Well, even with rule 07.04.01(04), compilers need to watch out for this situation and report errors if it occurs, e.g. the error in the package

```
package TEST_TYPE_LOOP is
        type T is private:
        type CONTAINS_T is record COMPONENT:T; end record;
private
        type T is record INDIRECTLY_ITSELF:CONTAINS_T;end record;
end TEST_TYPE_LOOP;
```

is flagged by the Alsys compiler as

```
                2 type T is private;
                          1
        1 **    SEM The type T is recursively dependent on another type.
```

So, type recursion checking doesn't become more complex if 07.04.01(04) would be discarded.

On the other hand, it is not clear which combination of rules of the reference manual should make this source not-compilable. If the manual does not contain such combination, this is at least an indication that the problem has not been properly dealt with.

Now, efficient loop checking is probably done by trying a topological sort on the type declaration dependencies, after replacing the private type declarations by the corresponding full declarations he could use to compile anything that can be sorted but does not satisfy rule 07.04.01(04). So, why putting restrictions on the programmer an additional rules in the reference manual when it is not necessary to do for software engineering reasons?

OVERLOADING OF "=" AND "/=" WITHOUT RESULT TYPE BOOLEAN

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 7.4.2 (3)

PROBLEM:

Currently, "=" and "/=" return a type BOOLEAN value where, for example for comparing arrays of values, it often is more appropriate and desirable to return an array (same index subtype) of BOOLEAN values or an Abstract Data Type (ADT) value containing more information.

For example, on a vector or parallel computer it is more efficient to compare all the elements of two vectors, returning a vector containing the individual comparisons - NOT a single BOOLEAN.

Further, the option to declare "=" is currently only allowed for new-declared limited types. We require the ability to issue declarations for "=" and "/=" for any operand type and result type.

IMPORTANCE: IMPORTANT

In particular for future Ada system implementations on vector computers and other high-performance novel architectures this facility is indispensable.

CURRENT WORKAROUNDS:

By differently named subprograms.

POSSIBLE SOLUTIONS:

To allow overloading and redeclaration of "=" and "/=" without restrictions on result type or non-limitedness.

DEFERRED CONSTANTS OF COMPOSITE INCOMPLETE TYPES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 7.4.3

PROBLEM:

Deferred constants of non-private types that have a subcomponent of a not-yet-completed private type should be allowed. For example, the following should be legal.

```
Package P is
        type T1 is private;
        Null_1 : constant T1;

        type T2 is private;
        Null_2 : constant T2;

        type R is
                record
                        F1 : T1;
                        F2 : T2;
                end record;

        Null_R : constant R;

private
        type T1 is new Integer;
        Null_1 : constant T1 := 0;

        type T2 is new Integer;
        Null_2 : constant T2 := null;
        Null_R : constant R := (F1 => Null_1, F2 => Null_2);
end P;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Declare a parameterless function instead of a constant. In addition to degrading performance, this will not work if the function is called before its body has been elaborated.

POSSIBLE SOLUTIONS:

Between the introduction and completion of a private type, allow the declaration of deferred constants of types that have a subcomponent of the private type.

DEFERRED CONSTANT PROBLEM

DATE: March 21, 1989

NAME: Bengt Sundelius

ADDRESS: ABB Automation
dep AUT/KMI
S-721 67 Vasteras
Sweden

TELEPHONE: +46 21 109254

ANSI/MIL-STD-1815A REFERENCE: 7.4.3 (4)

The execution of a program is erroneous if it attempts to use the value of a deferred constant before the elaboration of the corresponding full declaration.

PROBLEM:

The following situation exists.

```
package P is
 type T_TYPE is private;
 C : constant T_TYPE;

type R_TYPE is record
 X : T_TYPE := C;  --allowed
end record;

R : R_TYPE;          -- Here R.X is initialized
                     -- with the value of C
                     -- which is undefined.

private
 type T_TYPE is new INTEGER;
 C : constant T_TYPE := 0;
end P;
```

Further examples of a more complex nature can be produced from the above by adding discriminants to the private type. The general issue of erroneous execution and incorrect order dependence is raised in the revision request AdaUK 002.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Remove the object R from the specification.

POSSIBLE SOLUTIONS:

The example given above could perhaps be resolved by introducing a rule in Ada9X which would make it illegal. Alternatively, it may be possible to make the example legal with the expected semantics. We wish to avoid the possibility of erroneous execution since the actual effect of the program is then undefined (i.e. unpredictable).

ALLOW DEFERRED CONSTANTS OF ARBITRARY TYPES

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 7.4(4)

PROBLEM:

When a package must provide publicly accessable types, there is no way to provide publicly accessable constants without revealing the values of those constants.

IMPORTANCE: ESSENTIAL

Regularity is compromised without this, and the design goals of Steelman and of 1.3(3) are violated.

Concern for the human programmer was also stressed during the design. ... underlying concepts integrated in a consistent and systematic way. ...language constructs that correspond intuitively to what the users will normally expect.

CURRENT WORKAROUNDS:

Define a function that returns a constant value. This detracts from readability and hinders code optimization.

Define an otherwise unused private type and a transfer function to the underlying type. This is, if anything, worse than the first workaround.

POSSIBLE SOLUTIONS:

Allow deferred constants in a package specification, regardless of type.

As above, but also allow deferred initial values, i.e., duplicate declarations in a package specification and

the corresponding package body, differing only by the presence of an initial value in the package body. This would enable more information hiding.

OUT MODE PARAMETERS OF LIMITED PRIVATE TYPES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 7.4.4(4)

PROBLEM:

It should be legal to declare subprograms with OUT-mode parameters of limited types. For example, the following should be legal:

```
package P1 is
        type Lp1 is limited private;
        procedure Init (X : out Lp1);
private
        type Lp1 is new Natural;
end P1;

package body P1 is
        procedure Init (X : out Lp1) is
        begin
                X := ...;
        end Init;
end P1;

with P1;
package P2 is
        type Lp2 is limited private;
        procedure Init (X : out Lp2);
private
        type Lp2 is
                record
        F1 : FLoat;
        F2 : P1.Lp1;
end record;
end P2;

package body P1 is
        procedure Init (X : out Lp1) is
        begin
                X.F1 := ...;
                P1.Init (X.F2);
```

```
        end Init;
    end P2;
```

The current restrictions prevent this kind of composition of abstractions and thereby make limited private types more difficult to use effectively.

Section 6.2(6-7) should then be amended to require copy-in for OUT-mode parameters and parameter subcomponents of task types; this could be done by replacing the two occurrences of the work "access" in these two paragraphs with the text "access or task".

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Don't use limited types.

POSSIBLE SOLUTIONS:

PARAMETER MODES FOR LIMITED TYPES

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 7.4.4(4,9)

PROBLEM:

The rule that a formal parameter of a limited type can be of mode OUT only if the type is private and is declared in the package in which the procedure or entry in question is declared is unnecessarily restrictive. There is no reason why a procedure or entry in another package built on top of the one declaring the limited type should not be able to return such a parameter of mode OUT. If the original package provides an assignment operation for objects of the type, then the compiler can check to see that the assignment has been made in the body of the procedure or entry.

Likewise, the restriction that generic formal parameters of mode IN must not be of a limited type also prevents limited types from being used in a way that intuitively conforms with the spirit of the language. For instance, there is no reason why a default initial value cannot be passed to a generic unit as an IN parameter along with the type in question and an assignment operation to be used for initializing variables of the type.

The special cases concerning limited types, other than those involving assignment and equality, violate the general pattern of the language and discourage programmers from using limited private types. Limited private types should be made to behave exactly like private types, with the exception of assignment and equality. As it is, if a type were changed from a private to a limited private type, all OUT formal subprogram or entry parameters and all IN generic formal parameters of that type external to the package would have to be changed to IN OUT. But one of the purposes of using private types is to protect code external to a package from changing if the implementation of the type changes.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use IN OUT parameter modes with limited types.

POSSIBLE SOLUTIONS:

For additional references to Section 7. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 8. VISIBILITY RULES

LATE DEFINITION OF VISIBILITY RULES

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 8

PROBLEM:

All through the LRM, the reader finds forward references to scoping and visibility rules. The notes even provide examples for cases that the reader would never suspect from the rules that are given at that point in the language definition. The programmer has to reread the document several times. Most of the definitions are circular. The examples are so simplistic that not all of the ramifications are clear until a compilation system is actually used.

There is far too much vague wording and concepts being introduced here, e.g., library units. The scoping rules need to be expressed in terms of "external" or "export" and not words like "extends beyond immediate scope" as in 8.2 #2.

From the given scoping rules, a programmer would not deduce (f) in 8.3 #11.

Also, is it really necessary to introduce a semi-mathematical term such as "homograph" to introduce a common concept for scoping rules that most programmers already understand? Granted, conflicts in dynamic aspects can occur in Ada. For those cases, we would expect the compilation system to complain or warn. It is not clear what processing results when an overlap or an ambiguous object occurs. The error recognition should be at the earliest possible point of recognition and not merely raise an exception like PROGRAMMING_ERROR.

In note, 8.3 #22, the addition of the comment on the inner-most scope is not enough capability for a programmer to create a parameter and a surrounding procedure with the same name. I would expect that the overloading would take care of that. The compiler system should be able to tell the difference.

IMPORTANCE: IMPORTANT

Clarity and the scoping/visibility should be what an experienced programmer would expect.

CURRENT WORKAROUNDS:

Not much outside of many lines of code for declaring objects and for controlling scoping by very shallow nesting levels.

POSSIBLE SOLUTIONS:

Rewrite the LRM in a top-down axiomatic approach to developing the language. The "rationale" document should contain the BNF in a known grammar preprocessor such as YACC and LEX for the resultant grammar.

SIMPLIFY OVERLOADING FOR AMBIGUOUS/UNIVERSAL EXPRESSIONS

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 8

PROBLEM:

Currently the rules for overloading are unduly complex, especially in regard to implicit conversions of universal expressions. This means that in some circumstances, one interpretation is chosen over another based on the count of implicit conversions associated with the two interpretations, which is totally bizarre from a human understanding point of view. As things stand now, it also means that "for I in -1 .. N" doesn't work in certain contexts where "for I in 1 .. N" does.

Furthermore, there are cases where an otherwise ambiguous expression is considered unambiguous when there is only a single type declaration visible/accessible of a particular class. This means that expressions which are normally patently ambiguous (such as "null = null") are in certain unusual circumstances legal, creating confusion in the programmer's mind and unduly complex rules for the compiler.

IMPORTANCE: ESSENTIAL

The current rules for implicit conversion of universal expressions, and for disambiguating, represent one of the most confusing and painful-to-implement features of Ada.

CURRENT WORKAROUNDS:

There are no obvious workarounds.

POSSIBLE SOLUTIONS:

The simple solution for the implicit conversion problem is to defer implicit conversion from Universal type until absolutely necessary, and then perform the conversion on the universal expression, rather than on the primitives which make up the expression. The rules currently state that if implicit conversion is necessary, it must be applied to the "leaves" of the expression. This seems totally undesirable, and means that "1+1" doesn't always mean simply "2" depending on context. Furthermore, it means that "-1" (and +1) is not as good as "1" in some contexts.

A way to simplify the handling of ambiguous expressions is to state that expressions like "null", string literals, numeric literals, allocators, aggregates, etc., provide *no* information and their type must be fully

resolved from context. As it is now, a string literal may be assumed to be some array-of-character type, and if there is only one accessible with visible character numerals, then it is unambiguous. This requires the compiler to search *all* accessible scopes for such types under certain circumstances. Similar considerations apply to access types for allocators and null, and to composite types for aggregates. This means that when someone innocently introduces another type of this class in some accessible scope, such expressions suddenly become ambiguous. This seems highly undesirable. Numeric literals are different since they have a well-defined type, with well-defined operations, which is implicitly convertible to appropriate numeric types.

An alternative solution would be to consider all literals as some instance of a "universal" type, with operators if appropriate (e.g. "&" on string literals, "=" on null, etc), implicitly convertible when necessary to an appropriate compatible type. This means that literals have a well-defined type based strictly on their syntax, and that it would be legal to declare named literals of any sort without specifying a type.
For example:

```
blah : constant := "this is a universal " & "string literal";
```

Character literals would be similar. Non-character enumeration literals could not work this way since their syntax is not distinct from other identifiers, though it is easy to imagine an equivalence between a syntax like _red and red to allow for the declaration of "universal" numerals.

USE OF THE RENAMES BETWEEN THE SPECIFICATION AND BODY IN A PACKAGE

DATE: October 31, 1989

NAME: Gary McKee

ADDRESS: McKee Consulting
P.O. BOX 3009
Littleton, Colorado 80161
U.S.A.

TELEPHONE: (303) 795-7287

ANSI/MIL-STD-1815A REFERENCE: 8.1, 8.5

PROBLEM:

Currently, the RENAMES clause is not permitted to rename a subprogram in the specification of a package unless the RENAMES clause is also in the specification. This situation causes several problems. First, changes in the implementation of the package (by (1) renaming as a different subprogram; or, (2) producing code in the package body) requires recompilation of the specification and hence of all units that with that package.

Second, It is often desirable to hide the implementation details of a package for design reasons or for company proprietary purposes. Consequently, renames need to be in the package body where users of the package can't see them.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The best current workaround is to add another layer of indirection in the package body. This technique provides the needed functionality at the cost of increased code complexity, obfuscation of design precepts, and addition of an another layer of subprogram call overhead.

POSSIBLE SOLUTIONS:

The easiest answer, and one that more fully supports the engineering principle of information hiding, is to explicitly state that the RENAMES clause may be placed in the body of a package and still refer to the specification of a subprogram defined in the specification of that package. All other semantics of the RENAMES clause can be left the same.

NOTE: In my reading of the standard, I cannot conclusively prove that a RENAMES clause must be in the same declarative unit but tests run on six different compilers resulted in all six of them refusing the construct as incorrect. Perhaps all that is needed is an INTERPRETATION of the current standard that permits such a usage.

SELECTIVE VISIBILITY OF OPERATORS

DATE: July 21, 1989

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of Ada UK

ADDRESS: IPSYS plc,
Marlborough Court,
Pickford Street,
Macclesfield,
Cheshire SK11 6JD
United Kingdom

TELEPHONE: +44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 8.3

PROBLEM:

The Ada visibility mechanisms make it difficult to achieve selective direct visibility of declarations. This is particularly desirable in the case of predefined operators.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The first workaround is to achieve visibility by selection. For example, consider the simple package:

```
package ADT is
   type T is private;
   function "+" (LFFT,RIGHT:T) return T;
   procedure P (X; in T);
private
   --etc.
end ADT;
```

and use of the subprograms provided as follows:

```
with ADT;
procedure USE_ADT is
   A,B,C : ADT.T;
begin
   if ADT."=" (A,ADT."+"(B,C)) then
      ADT.P (A);
      --etc.
end USE_ADT;
```

Selection is usually acceptable and often preferable for procedures, but as can be seen is clumsy for

operators.

Another workaround is to achieve direct visibility through use clauses. For example:

```
with ADT;
procedure USE_ADT is
    use ADT;
    A,B, C : ADT.T;
begin
    if A = B + C then
        ADT.P (A);
        -- etc.
end USE_ADT;
```

Here, however, all of the declarations in package ADT are made directly visible.

Yet another workaround is to systematically rename the declarations whose direct visibility is required, e.g.:

```
with ADT;
procedure USE_ADT is
  A, B, C: ADT.T
  function "+"(LEFT,RIGHT : ADT.T) return ADT.T renames ADT."+";
  function "="(LEFT,RIGHT : ADT.T) return BOOLEAN renames ADT.>"=";
  --as required
begin
  if A = B + C then
      ADT.P (A);
      --etc.
end USE_ADT;
```

This workaround can be laborious and error prone without automated assistance.

POSSIBLE SOLUTIONS:

One solution would be to allow more selective use clauses, e.g.:

```
use ADT."+";
use ADT."=";
--etc.
```

or perhaps:

```
use ADT.<>;
```

which would make all operators from ADT directly visible. Another form of use clause could make all overloadable entries directly visible, but not other entities.

Another solution would be for the predefined operations, at least, of a type to have direct visibility in expressions involving objects of that type. This is a very natural expectation which currently applies in the case of assignment.

SELECTIVE IMPORTATION OF TYPE OPERATIONS

DATE: October 21, 1989

NAME: Gary Dismukes

ADDRESS: TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9891

TELEPHONE: (619) 457-2700 x322

ANSI/MIL-STD-1815A REFERENCE: 8.3, 8.4

PROBLEM:

There is not a convenient, concise way of providing direct visibility of operations such as enumeration literals and predefined operators belonging to a type declared in a package without introducing visibility of the entire package interface. The workarounds for this problem serve to hinder the readability and maintainability of Ada programs.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

1) Use clauses can be used to provide direct visibility of operations, but at the cost of polluting the name space of scopes. This can have the consequence of leading programmers into a style of always using use clauses to achieve visibility.

2) Selective notation can be used to invoke operators and reference enumeration literals. This results in a verbose style that hurts readability of programs.

3) Literals and operators can be renamed within scopes requiring their use. This workaround provides the greatest degree of control over importation, but is verbose and inconvenient when many literals or operators need to be imported.

POSSIBLE SOLUTIONS:

Ideally one would like a concise syntax for easily importing arbitrary symbols or related sets of symbols from a given package by specifying only the simple name of the symbol or a special type attribute in the case of sets like operators or literals. For example:

```
From Pkg import X, T, Func, T'operators, T'literals;
```

This may import too much, however, as for overloaded subprograms, in which case a renaming declaration can be used. The visibility here would be as if a renaming declaration had occurred for each of the named entities (unlike the visibility introduced by a use clause). Unfortunately, introducing the above syntax would probably not be feasible because it violates upward compatibility by requiring new reserved words.

Alternatively, the semantics of use clauses could be extended, to permit importation of operations associated with a specific type. If the name given in a use clause were a type (or a first-named subtype), then direct visibility would be enabled for any enumeration literals or predefined operations of the base type. It would be nice to be able to distinguish importation of operators from importation of enumeration literals using an attribute notation, but importation of all type operations would by itself go a long way towards improving the readability and maintainability of programs written using this construct in preference to straight use clauses.

It would be preferable for the visibility introduced by a "type" form of use clause to be as if renaming declarations for the various operations were introduced at the point of the use clause, however this would be inconsistent with the current semantics of use clauses. Thus, visibility would probably need to be defined as if an implicit package had been introduced (containing renamings of the operations) followed by an associated use clause for the implicit package.

Potential design difficulties in adding such a feature include questions of visibility interference due to use clauses for the package defining the type and issues of what to do if the package defining the type contains superseding homographs for the "used" type's literals or operators.

SIMPLER USE CLAUSE VISIBILITY RULES

DATE: October 16, 1989

NAME: Samuel Mize

DISCLAIMER:

The views expressed are those of the author, and do not necessarily represent those of Rockwell International.

ADDRESS: Rockwell/CDC
3200 E. Renner Road M/S 460-220
Richardson, TX 75081

TELEPHONE: (214) 705-1941

ANSI/MIL-STD-1815A REFERENCE: 8.3, 8.4,

PROBLEM:

The USE statement confuses a lot of people.

According to G. O. Mendal [1], "From my experience, only one out of five programmers fully comprehends the intricate semantics of the use clause. The remaining 80% ... believe that the use clause achieves direct visibility simply by its inclusion in a program."

For example, an identifier made visible by a USE clause can be hidden by another identifier further out in the nested scopes around it. To add an identifier to a package inside a program, a programmer must (manually or with CASE tools) check for usage of that identifier, not only within the package declarations's scope, but in all scopes around the package.

The Ada Rationale [2] explains that these visibility rules were selected to prevent maintenance problems from changes to packages. However, Mendal gives scenarios that show how a USE statement with the current visibility rules can also cause errors in maintenance.

IMPORTANCE: IMPORTANT

A clearer USE clause semantics would encourage use of the USE clause, providing more readable and maintainable code. The USE clause is now completely avoided by one segment of the Ada community, and misused by another (perhaps larger) segment.

However, existing code may not operate correctly with the new USE semantics, unless a different keyword is used to differentiate the new semantics from the old.

CURRENT WORKAROUNDS:

1. Provide specific training and CASE tools that allow correct use of the USE clause in large programs.

2. Forbid use of the USE clause (as Mendal recommends)

POSSIBLE SOLUTIONS:

Simplify the semantics of the USE clause by deleting the LRM 8.4 the following exception to direct visibility from a USE clause:

> "A potentially visible declaration is not made directly visible if the place considered is within the immediate scope of a homograph of the declaration."

This would allow a declaration that is made directly visible by a USE clause to hide an otherwise directly visible declaration. It would, in effect, make declarations in a USEd package as directly visible as if they were directly declared in the scope in which the package is declared. This is a more intuitive and understandable behavior.

THe existing USE semantics could be retained, with the new behavior attached to a new keyword or sequence of keywords in the USE statement. This would maintain upward compatibility. For example, "USE package_1" would work as it does now; "USE ALL OF package_1" would make all of package_1's identifiers directly visible as if directly declared where the package is declared.

REFERENCES:

[1] Mendal, Geoffrey O. "Three Reasons to Avoid the Use Clause." Ada Letters, volume VIII, no. 1, Jan/Feb 1988.

[2] Ichbiah, Jean D.; Barnes, John G. P.; Firth, Robert J.; Woodger, Mike. Rationale for the Design of the Ada Programming Language. Honeywell Systems and Research Center, 1986. Section 11.3.2.

WRAPPING A LIBRARY UNIT'S DECLARATIVE SCOPE AROUND A COMPILATION UNIT, BY MEANS OF A CONTEXT CLAUSE

DATE: October 16, 1989

NAME: Samuel Mize

ADDRESS: Rockwell/CDC
3200 East Renner Road
M/S 460-220
Richardson, TX 75081

TELEPHONE: (214) 705-1941

ANSI/MIL-STD-1815A REFERENCE: 8.3, 8.4

The views expressed are those of the author, and do not necessarily represent those of Rockwell International.

PROBLEM:

The USE statement confuses a lot of people.

According to G.O. Mendal [1], "From my experience, only one out of five programmers fully comprehends the intricate semantics of the use clause. The remaining 80% ... believe that the use clause achieves direct visibility simply by its inclusion in a program."

A USE clause only creates potential visibility into a package; this visibility can change based on changes to any scope enclosing the USE statement's region of effect (e.g., adding an identifier identical to one used in the referenced package). Such changes can affect this visibility whether or not the USEing unit intentionally uses the declarations from the enclosing scope.

This is especially a problem with separated subunits, where the program test is in several (often quite a few) files. One would like to be able to code a separate subunit based on its own identifiers and those of any USEd packages. One would like to not have to know about identifiers in the enclosing scopes, except for any that the subunit is using. This is the case with identifiers directly declared in the subunit, but not for identifiers in packages it USEs.

Maintenance of a subunit's enclosing scope should not affect the subunit, except for where the subunit actually uses a changed item from the enclosing scope.

CONSEQUENCES OF THE PROBLEM:

Mendal[1] gives scenarios that show how a misunderstood USE statement can cause erroneous programs. Racine[2] argues that such problems arise from bad Ada style; even if this is true, bad style will no doubt continue to exist. The appended examples (at the end of the file) show some of the problems in brief.

IMPORTANCE: IMPORTANT

The USE clause is now completely avoided by one segment of the Ada community, and misused by another

(perhaps larger) segment.

CURRENT WORKAROUNDS:

1. Use specially trained programmers and/or CASE tools. Always check for name conflicts in all scopes enclosing every subunit that USEs a package whenever the USEd package is changed. Always check for name conflicts in all packages USEd anywhere within a scope if an identifier is added to the scope.

2. Don't use USE clauses.

POSSIBLE SOLUTIONS:

This revision request does not propose changing the meaning of any currently-legal program.

I propose adding new context-clause syntax that makes a WITHed package's identifiers visible as if declared in a new scope directly enclosing the compilation unit.

This would be usable only in context clauses, not in a USE clause within a declarative region. That would have much greater potential for creating the problem referred to in the Ada Rationale[3], 11.3.2, eighth text paragraph: "In particular, the inner reference to X should retain its previous meaning and should hence mean Q.X both before and after the modification."

One usable syntax would be "WITH [DECLARE OF] <package_name>". A context clause that used the DECLARE OF keywords would not require (and could not accept) a USE clause naming the same package.

PROBLEMS OF PROPOSED SOLUTION:

Adding another kind of context clause would increase the complexity of reading an Ada program. Programmers would have to be sure to use the type of context clause they intended, and to read context clauses carefully when maintaining software.

BENEFITS OF PROPOSED SOLUTION:

This proposal would allow a kind of declaration importation that many programmers are familiar with from other languages (e.g., Turbo Pascal). This importation is simpler to understand and easier to use correctly.

Adding this type of declaration importation would also make new Ada users aware that the existing WITH/USE combination does NOT work as they might expect. This would reduce USE clause mistakes.

Programmers, now sure about the side effects of their actions, will be more inclined to correctly use both the new construct and the existing WITH/USE clause, promoting more readable code.

EXAMPLES:

EXAMPLE 1

```
package External_Pkg is
        procedure Do_Something;
end External_Pkg;
package body External_Pkg ...
```

```
-------------------------------------------------------------------------------
procedure Pgm is ... -- dashed lines show separate compilation units
-------------------------------------------------------------------------------
with External_Pkg; use External_Pkg;
separate (Pgm.A.B.C.D.E.F.G)
procedure Too_Deeply_Nested is begin
        Do_Something;
        -- is this the Do_Something from External_Pkg, or
        -- another from some higher enclosing scope?
end Too_Deeply_Nested;
```

EXAMPLE 2

```
--External_Pkg is a solid, tested, reliable unit.
package External_Pkg is
        C : integer;
end External_Pkg;
-------------------------------------------------------------------------
--Pgm is in work, and therefore may often change.
program Pgm.
        A : integer;

        package Troublemaker is
                D : integer;
        end Troublemaker;
        use Troublemaker;

        package Contained_Pkg is
                function CPD return integer;
        end Contained_pkg;
        package body Contained_Pkg is separate;

        procedure Trouble1 is begin ... D := 1 ... end;

begin end;
-------------------------------------------------------------------------
with External_Pkg; use External_Pkg;
package body Contained_Pkg is
        D : integer;
        function CPD return integer is
                begin return D; end;
begin
        D := A + B + C;
end;
```

COMMENTS ON EXAMPLE 2

With current Ada semantics, changes to the code have the following effects:

ADDING	TO	DOES	AND FORCES RECOMPILE OF	
A	External_Pkg	nothing	External_Pkg Contained_Pkg	
B	Pgm	CHANGES MEANING OF CONTAINED_PKG	Pgm Contained_Pkg	*1
C	Troublemaker	makes Contained_Pkg illegal	Pgm Contained_Pkg	*2
D	External_Pkg	nothing	Contained_Pkg	
D	Troublemaker or Pgm	nothing	Pgm Contained_Pkg	

*1 recoding is necessary for correctness, NOT FORCED BY LANGUAGE
*2 recoding required by compiler

Replacing the context clause on package body Contained_Pkg with the proposed WITH DECLARE OF context clause, changes to the code will have the following effects-

ADDING	TO	DOES	AND FORCES RECOMPILE OF	
A	External_Pkg	CHANGES MEANING OF CONTAINED_PKG	Contained_Pkg	*1
B	Pgm	nothing	Pgm Contained_Pkg	
C	Troublemaker	nothing	Pgm Contained_Pkg	
D	External_Pkg	nothing	Contained_Pkg	
D	Troublemaker or Pgm	nothing	Pgm Contained_Pkg	

*1 recoding is necessary for correctness,
NOT FORCED BY LANGUAGE

*1 recoding is necessary for correctness, NOT FORCED BY LANGUAGE

Table 2 reflects more the behavior that the programmer of the separate package Contained_Pkg would prefer.

REFERENCES:

[1] Mendal, Geoffrey O. "Three Reasons to Avoid the Use Clause." Ada Letters, volume VIII, no. 1, Jan/Feb 1988.

[2] Racine, Roger, "Why the Use Clause is Beneficial," Ada Letters, volume VIII, no. 3, May/June 1988.

[3] Ichbiah, Jean D.; Barnes, John G. P.; Firth, Robert J.; Woodger, Mike. Rationale for the Design of the Ada Programming Language. Honeywell Systems and Research Center, 1986.

THE PARAMETER AND RESULT TYPE PROFILE OF OVERLOADED SUBPROGRAMS

DATE: October 11, 1989

NAME: Mats Weber
Endorsed by Ada-Europe, number AE-021,
originator : Stef Van Vlierberghe

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail : madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 8.3(15), 6.6

PROBLEM:

Subprograms with the same parameter and result type profile, but different formal parameter names, are currently allowed only if they are not declared in the same declarative region. This seems inconsistent with the fact that the overloading rules can differentiate calls to such subprograms if the named notation is used for parameter passing.

For example, the following package declaration is illegal:

```
package Illegal is

   type Text is ...;

   function Substring (From : Text; First : Positive; Last   :   Natural);
   function Substring (From : Text; First : Positive; Length :   Natural);

      -- Note that the current overloading resolution rules can
      -- differentiate the two subprograms if named parameter
      -- passing notation is used.

end Illegal;
```

but the following is legal

```
package Legal is

   type Text is ...;

   package Substring_Function_1 is
      function Substring (From : Text; First : Positive; Last : Natural);
   end;
```

```
    package Substring_Function_2 is
       function Substring (From : Text; First : Positive; Length : Natural);
    end;

end Legal;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use different names for the subprograms, or declare them in subpackages as in the above example package Legal.

POSSIBLE SOLUTIONS:

Include the parameter names in the parameter and result profile. Note that his may cause problems with renaming declarations.

PROVIDE CONSISTENT VISIBILITY OF GENERIC SUBPROGRAMS AND PACKAGES

DATE: October 23, 1989

NAME: Allan R. Klumpp

ADDRESS: Jet Propulsion Laboratory
4800 Oak Grove Drive
Mail Stop 301-125L
Pasadena, CAL 91109

TELEPHONE: 818-354-3892
FTS 792-3892
Internet:KLUMPP@JPLGP.JPL.NASA.GOV
Telemail:KLUMPP/J.P.L.

ANSI/MIL-STD-1815A REFERENCE: 8.3.16

ALIWG ACTION: Favorable vote 1987 in Boston and 1988 in Charleston, W.V.

PROBLEM:

The visibility rule of LRM section 8.3 paragraph 16 makes it impossible to instantiate a generic subprogram without changing the name, although a name change is often undesirable. The same rule would mandate a name change when using the proposed deriving declaration (LI42). The rule does not apply to packages, an unnecessary inconsistency.

Example

```
PACKAGE VISIBILITY_RULES IS

    GENERIC -- Function PROD

        TYPE INTEGER_G IS RANGE <>;

    FUNCTION PROD (LEFT: INTEGER_G; RIGHT: INTEGER_G) RETURN
INTEGER_G;

    GENERIC -- Package NESTED

        TYPE FLOAT_G IS DIGITS <>;

    PACKAGE NESTED IS

        . . . ;

    END NESTED; -- End Spec

END VISIBILITY_RULES; -- End Spec
```

```
WITH VISIBILITY_RULES; USE VISIBILITY_RULES;
PROCEDURE VISIBILITY_RULES_TEST IS

    FUNCTION PROD IS NEW VISIBILITY_RULES.PROD
        (INTEGER_G => INTEGER);          --Does not compile
    PACKAGE NESTED IS NEW VISIBILITY_RULES.NESTED
        (FLOAT_G => LONG_FLOAT);         --Does compile

END VISIBILITY_RULES_TEST; -- End Procedure
```

The visibility rule prevents instantiating the function but not the package.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

There are no workarounds. The consequence is more cumbersome code.

POSSIBLE SOLUTIONS:

The governing rule, LRM section 8.3 paragraph 16, reads:

Within the specification of a subprogram, every declaration with the same designator as the subprogram is hidden; the same holds within a generic instantiation that declares a subprogram, and within an entry declaration or the formal part of an accept statement; where hidden in this manner, a declaration is visible neither by selection nor directly.

The purpose of this rule is to make it impossible to declare a subprogram recursively,i.e., in terms of itself, but, for the sake of simplicity, the rule is more general than necessary.The unneeded generality causes the problem.

Here is the proposed replacement for the preceding rule:

Within the specification of a subprogram or a package, the declaration of the subprogram or package itself is hidden; the same holds with in a generic instantiation that declares a subprogram or a package, and within an entry declaration or the formal part of an accept statement. Where hidden in this manner, the declaration is visible neither by selection nor directly.

There placement achieves the objective of the current rule and eliminates the undesirable side effects. With the proposed rule, subprograms cannot be declared recursively, but generic subprograms and packages can be instantiated without changing their names. Furthermore, it seems to satisfy the objections to the current rule that were raised in 1982 (see Ada Comments from the Ada Information Clearinghouse).

RELAX VISIBILITY RESTRICTIONS WITHIN SUBPROGRAM SPECIFICATIONS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 8.3(16)

PROBLEM:

It would be useful if the following procedure declaration were legal:

```
package P1 is
        function Foo return Integer;
end P1;

package P2 is
        type Foo is ...;
end P2;

procedure Foo (X : Integer := P1.Foo; Y : P2.Foo);
```

The restrictions given in 8.3(16) seem very unintuitive, particularly since declarations such as

```
package Q1 is
        subtype X is Integer;
end Q1;

package Q2 is
        X : Integer := 1;
end Q2;

X : Q1.X := Q2.X;
```

are legal.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Completely remove, or at least relax, the restrictions given in 8.3(16).

USE OF A NAME IN ITS DEFINITION

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 8.3, 16

PROBLEM:

The reference manual says that a name cannot be used in the definition of a subprogram of that name (or a generic instantiation that declares a subprogram of that name), even if it does not refer to the subprogram being defined. For example, if I have a package Avoirdupois containing a type Pound, and I want to instantiate the generic procedure Bang_On:

```
generic
        type Pressure;
procedure Bang_on is
begin
        null;
end bang_on;

with avoirdupois;
with bang_on;
package Whatever is
        procedure Pound is new bang_on(avoirdupois.pound);
end whatever;
```

This is illegal because the use of the token pound is forbidden within the definition of pound. This restriction is overly broad.

This also reflects an inconsistency in the handling of names in the language:

```
with text_io;
package User is
function Mode(File: text_IO.file_type)
            Return text_IO.mode; -- illegal
function Name(File: text_IO.file_type)
            return string
            renames text_IO.name; -- legal
end user;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Either call the local procedure something else, or use a temporary pseudonym:

```
with avoirdupois;
with bang_on;
package Whatever is
        procedure Ezra is new bang_on(avoirdupois.pound);
        procedure pound renames Ezra;
end whatever;
```

This is unsatisfactory, in that an extraneous name (Ezra) is exported.

POSSIBLE SOLUTIONS:

Restrict the restriction so that it forbids only those uses of the name which either (a) refer to the thing being defined or (b) use a form of reference which will not be correct after the definition is complete. Since the analysis of condition (b) is complex, a reasonable stronger condition is to forbid any use which could be interpreted to refer to a declaration of the name in the current scope. This prevents any need to disambiguate a name reference which may be overloaded with an incomplete symbol entry.

INVISIBLE IMPORTED TYPES IN SUBPROGRAM FORMAL PARTS

DATE: October 20, 1989

NAME: Bryce M Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 8.3(17)

PROBLEM:

The following anomaly exists in Ada83, and is encountered with significant frequency by end users.

Usually package P already exists (it is reusable, or written by another organization).

```
package P is
        type T is...
end P;

-- Attempt to define procedure T.
with P; {use P;}
procedure T(X : P.T); -- P.T not visible by selection
                      -- T is not visible when the use clause is
                      -- present either
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

1) The existing package may be modified, but this is often undesirable and may be under the programmer's control:

```
package P is
        type T is...
        subtype U is T;
end P;

with P;
procedure T(X : P.U);
```

2) The procedure name may be abandoned (but the user sees this as an unnecessary quirk in the language):

```
package P is
        type T is ...
end P;

with P;
procedure U(X : P.T);
```

POSSIBLE SOLUTIONS:

However, it seems harmless to the user at least, to allow the visibility of P.T by selection in this case:

```
package P is
        type T is ...
end P;

with P;
procedure T(X : P.T); -- allow visibility of P.T by selection
```

CONVERSIONS

DATE: October 20, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 8.3(17), 8.3(5,22)

PROBLEM:

Ada is non_symmetric with respect to conversion functions. The names of numeric types can be used as conversions functions, but not the names of other types. For any type, it should be possible to use the name of a type as a type transfer function. Function names should not hide type names and vice versa.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Use different function names. For example,

```
package Thing_manager is
        type Thing_type is private;
        function Thing_type( Number: integer ) return  Thing_type;
        function integer( Thing: Thing_type ) return integer;
        procedure Process( Thing: in out Thing_type );
private
        type Thing_type is record
                N : integer;
        end record;
end Thing_manager;
```

would have to be rewritten with different names for the functions.

POSSIBLE SOLUTIONS:

It should be possible for the user to write

```
X := integer( Y );
A := Thing_type( B );
```

for any integer X and any object Y which is non-numeric, provided that a conversion function has been created. Also for any object B, provided that an appropriate Thing_type function has been declared, B should be convertible to A by means of a user-defined type.

There should be no need to treat numeric types differently from user-defined types.

As an added example, a varying strings package should be able to export:

```
package Varying_strings is
        type Varying( Maximum_length: positive ) is private;
        function Varying( Item: string  ) return Varying;
        function String ( Item: Varying ) return string;
private
        ...
end Varying_strings;
```

OVERLOADING OF GENERIC SUBPROGRAM NAMES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 8.3(21)

PROBLEM:

Overloaded generic subprograms are not permitted by the standard, even when there is enough information in their parameter profiles to disambiguate them, as would be done with the non-generic overloaded subprograms. For example, this will not compile (the Second Bar will be flagged as a co-resident homograph of the first):

```
generic
        type Foo is private
procedure Bar (Mumble : in Integer);

generic
        type Foo is private
procedure Bar (Mumble : in Boolean);
```

This forces the programmer to use a less-satisfactory name for one of the overloaded generic subprograms, which subverts the cause of object-oriented programming (since there is often a single ideal name for both, such as Put).

IMPORTANCE: ESSENTIAL

It makes no sense for the standard to enforce a stricter set of rules for disambiguating generic subprograms than for non-generic subprograms, especially when the amount of information that could be used for disambiguation is actually greater in the case of generics (since the generic formals region could also conceivably used to disambiguate at the point of instantiation); this asymmetry is particularly annoying because it thwarts the symmetrical programming style used in object-oriented programming.

CURRENT WORKAROUNDS:

Use a less-satisfactory name for one of the overloaded generic subprograms.

POSSIBLE SOLUTIONS:

Modify the standard so that the rules for disambiguating overloaded generic subprograms are the same as the rules for disambiguating overloaded non-generic subprograms.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

REQUIREMENT TO ALLOW PARTITIONING OF ADA PROGRAMS OVER MULTIPLE PROCESSORS IN DISTRIBUTED/PARALLEL/MULTI-PROCESSOR ENVIRONMENT

DATE: August 25, 1989

NAME: V. Ohnjec (Canadian AWG #013)

ADDRESS: 240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 8.4, 13.x.x, Annex C

PROBLEM:

Ada does not allow partitioning of programs over a multiple processor environment.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Multi-linguistic and specific vendor run-time systems/design solutions can be developed.

POSSIBLE SOLUTIONS:

Enhance Ada language to allow for specific choice of what packages go where, within the multi-processor environment.

Requires additional discussion by Ada 9X revision members.

Impacts are potentially great for compiler vendors (for linker development).

NEED USE CLAUSE FOR OVERLOADABLES ONLY

DATE: October 18, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 8.4

PROBLEM:

An additional kind of "use" clause should be provided which provides direct visibility to overloadable entities, without providing direct visibility to non-overloadable entities within a package.

Use clauses have been frequently criticized because they can complicate the process of reading and understanding a piece of code. The most useful aspect of use clauses is that they allow direct use of operators, enumeration literals, and string literals (if appropriate) for a type defined within a separate package. Each of these uses are easier to understand because they are operations rather than types or objects, and hence never will stand alone without some context which gives the reader a hint of their meaning.

Non-overloadable entities made visible by use clauses can cause great confusion to a reader. An object name, or worse, a type name, can be used essentially in a stand-alone manner, and the reader has no clue as to the point of definition of the entity.
Another confusing aspect of use clauses is that a non-overloadable entity cancels out all other entities with the same identifier made visible by use clauses.

Given these issues, it seems "useful" to provide a variant of the use clause which only makes non-overloadables visible.

IMPORTANCE: IMPORTANT

If alternatives to the current use clause are not provided, the use clause will remain a denigrated feature of the language, and it will continue to produce tirades in the Ada literature concerning its proper and improper use, etc.

Ada code sprinkled with use clauses will continue to be very difficult to comprehend.

CURRENT WORKAROUNDS:

To only gain visibility to overloadable entities, a (tedious) sequence of explicit renames may be inserted. This creates maintenance problems due to the required duplication of subprogram specs, etc., and doesn't help for string literals.

Alternatively, the use clause may be inserted, and then the programmer can be careful not to reference non-overloadables without prefixing them with a package name (or rename).

POSSIBLE SOLUTIONS:

I can imagine two possible solutions. The first, is to create a special use clause which makes all non-overloadables of a package visible. For example:

use'overloadables <package-name>;

However, I much prefer the following alternative:

use <type-mark>;

This kind of use clause would be defined to make directly visible the subprograms and operations which would be implicitly declared as part of a type derivation from <type-mark>.

This has several advantages:

-- Only overloadable entities are made directly visible by this use clause.

-- Subtypes gain a more favorable status, because they may be "use"d to gain visibility to their operators, whereas now a package defining a subtype, when "use"d, does not provide visibility to the associated operators.

LIMITED USE CLAUSE FOR DIRECT VISIBILITY TO OPERATORS

DATE: October 23, 1989

NAME: Mike Glasgow

ADDRESS: IBM Systems Integration Division
9231 Corporate Blvd.
Rockville, MD 20850

TELEPHONE: (301) 640-2834

ANSI/MIL-STD-1815A REFERENCE: 8.4

PROBLEM:

The current standard does not provide an efficient means for achieving direct visibility to operators in a WITHed package without at the same time providing direct visibility to all other declarations in the package specification (private part and homographs excluded).

The 'ultimate' appropriate use of the USE clause is recognized to be a debatable issue. However, one reasonable and arguable position is that the USE clause should be avoided as much as possible to make code more readable and maintainable. Full qualification of types, subtypes, objects, enumeration literals, subprograms, and task entries is easily accepted (particularly with the use of RENAMES) by programmers because readability remains natural and eventually they discover that the qualification is to their benefit. However, readability becomes unnatural where operators are involved. For example,

```
x := (i + j)/k;
```

is without question more readable than

```
x:= package_name."/"(package.name,"+"(i, j), k);
```

However, with current 'all or nothing' approach to the USE clause, one must choose between undesired direct visibility to other declarations or adding renaming declarations in one of at least two ways (discussed in workaround section below).

IMPORTANCE: IMPORTANT

The net effect of not providing an efficient means for achieving direct visibility to only operators is that applications will continue to be developed that are less maintainable than they otherwise could be due to:

1. Reduced readability of algorithmic code
2. Reduced readability as a result of not being able to quickly resolve external references
3. Increased source lines of code (SLOC)

CURRENT WORKAROUNDS: Besides using the USE clause (either in a context clause or in BLOCK statements to limit the scope (awkward coding style)), there are at least 3 workarounds.

1. Do derived type declarations for types for which visibility to operators is desired and use type conversions. This leads to less readable code as well as due to the additional type conversions, though this isn't quite as bad as qualified operators.

2. Code local renaming declaration for all operators for which direct visibility is desired. This leads to excessive repetition of code and inflates SLOC counts.

3. (Workaround being used on FAA Advanced Automation System program et al.) In the package where a type is defined, create a nested package that contains renames for all the operators associated with the type. Then clients can WITH the library package and USE the inner package to get visibility just to the operators (or whatever one chooses to place in the inner package).

   Example:

```
package COLORS is
type BASE_COLORS is (RED, GREEN, BLUE);
        package OPS is
        function "=" (L,R : in BASE_COLORS) return BOOLEAN
                renames COLORS"=";
        ...
        end OPS;
end COLORS;
```

   The benefit of this workaround is that the renaming declarations are coded only once instead of being duplicated in every client. It can also be tooled. However it still increases SLOC counts and raises the maintenance coat.

POSSIBLE SOLUTIONS:

Two possibilities:

1. Provide a LIMITED USE (or USE LIMITED) construct that achieves the effect of the current USE clause only for operators. (It is not intended that this construct replace the current USE clause.) This construct could easily be implemented such that previously existing code is upwards-compatible with Ada 9X.

2. (Preferable solution) Provide a capability for selective visibility, similar to that provided by Modula-2. In addition to being able to specify specific entities for direct visibility, provide support for identifying classes of entities, such as operators. (Other possible classes: enumeration literals, functions, attributes, types/subtypes) This could also be implemented such that previously existing code is upwards-compatible.

COMPONENT-SPECIFIC CLAUSES

DATE: October 31, 1989

NAME: Gerald L. Mohnkern

ADDRESS: DARPA/ISTO
1400 Wilson Blvd
Arlington, VA 22209

TELEPHONE: (703) 522-2371
E-mail: mohnkern@nosc.mil

ANSI/MIL-STD-1815A REFERENCE: 8.4

PROBLEM:

Neither Ada's "with" or "use" clauses indicate which components of the "withed" package are used in the current program.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Some programmers never use "use" clauses, so that they always use the *full "dotted"* name of the component. Others use "use" clauses for all "withed" packages and leave it to the reader to determine the source package for each component. Others (including me) prefer to use "use" clauses, but comment each with the names of the components that come from that package.

POSSIBLE SOLUTIONS:

Component specific "use" clauses should be added to the language, akin to the "FROM...IMPORT..." statement of Modula-2.

VISIBILITY OF ARITHMETIC OPERATIONS

DATE: September 29, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Work: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Work: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: 8.4(2), C(11)

PROBLEM:

Basic arithmetic operations on a fixed point type, defined in a package specification, are not directly visible in another module that imports that package. Some of them can be made visible with a renaming declaration in the caller, but that will not work for divide, since it returns a value of universal-fixed.

IMPORTANCE: IMPORTANT (See also 0022.)

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

1. Allow a context clause to specify a fully qualified type, function or procedure name, e.g.,

   use MY_PACKAGE.MY_FIXED_TYPE, MY_PACKAGE."/";

   In case of overloading, all matching declarations should be made visible.

2. Change the definitions of arithmetic operations on fixed point types so that all operators yield a result of a specific type. This is a rather drastic change, with lots of side effects, and I would not recommend it.

3. Allow a specification of return STANDARD.UNIVERSAL_FIXED in a renaming declaration. This is ugly, but would do the job with minimal changes to the language.

RENAME SUBPROGRAM BODIES

DATE: July 25, 1989

NAME: R. David Pogge

ADDRESS: Naval Weapons Center
EWTES - Code 6441
China Lake, CA 93555

TELEPHONE: (619) 939-3571
Autovon: 437-3571

ANSI/MIL-STD-1815A REFERENCE: 8.5

PROBLEM:

Often it is desirable to declare subprograms in a package specification and implement them with renamed programs in the package body. For example, consider the VIRTUAL_TERMINAL package specification from page 295 of "Ada in Action" by Do-While Jones.

```
package VIRTUAL_TERMINAL is
   -- lots of declarations including...
   function Keyboard_Data_Available return boolean;
end Virtual_Terminal;
```

There are multiple bodies for this terminal that are system dependent. We would like to write them this way:

```
with DOS; --Alsys environment package
package body VIRTUAL_TERMINAL is
   -- lots of bodies including...
   function Keyboard_Data_Available return boolean
     renames DOS.Kbd_Data_Available;
end VIRTUAL_TERMINAL;

with TTY; -- Meridian environment package
package body VIRTUAL_TERMINAL is
   -- lots of bodies including ...
   function Keyboard_Data_Available return boolean
     renames TTY.Char_Ready;
end VIRTUAL_TERMINAL;
```

IMPORTANCE: IMPORTANT

This is IMPORTANT because it makes programs cleaner, but not essential because there are acceptable workarounds.

CURRENT WORKAROUNDS:

Workaround 1. Make implementation specific package specifications. That is,

```
with DOS; -- Alsys environment package
package VIRTUAL_TERMINAL is
   --lots of specifications including...
   function Keyboard_Data_Available return boolean
      renames DOS.Kbd_Data_Available;
end VIRTUAL_TERMINAL

with TTY; -- Meridian environment package
package VIRTUAL_TERMINAL is
   --lots of specifications including ...
   function Keyboard_Data_Available return boolean
     renames TTY.Char_Ready;
end VIRTUAL_TERMINAL;
```

The philosophical problem with this workaround is that it makes implementation details (the use of DOS and TTY) visible in the package specification. The practical problem with this is that it forces recompilation of dependent units when implementation details change.

Workaround 2. (preferred) Nest bodies in bodies.

```
package VIRTUAL_TERMINAL is
   -- lots of declarations including ...
   function Keyboard_Data_Available return boolean;
end VIRTUAL_TERMINAL;

with DOS; -- Alsys environment package
package body VIRTUAL_TERMINAL is
   -- lots of bodies including ...
   function keyboard_Data_Available return boolean is
   begin
     return DOS.Kbd_Data_Available;
   end Keyboard_Data_Available;
end VIRTUAL_TERMINAl;

with TTY; -- Meridian environment package
package body VIRTUAL_TERMINAL is
   -- lots of bodies including ...
   function Keyboard_Data_Available return boolean is
   begin
     return TTY.Char_Ready;
   end Keyboard_Data_Available;
end VIRTUAL_TERMINAL;
```

The disadvantages with this workaround are:

(1) The optimizer won't replace the double function call with a single function call to the desired

subprogram, resulting in slower execution speed.

or

(2) The optimizer will replace the double function call with a single function call to the desired subprogram, resulting in slower compilation speed.

POSSIBLE SOLUTIONS:

Add the following option to the list of renaming declarations:

```
renaming_declaration ::=
   (other options as before)
   | subprogram_body renames subprogram_or_entry_name;
```

DIFFICULTIES WHEN A LIBRARY UNIT IMPORTS TYPE DECLARATIONS FROM ELSEWHERE

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 8.5, 3.3.2

PROBLEM:

There are difficulties when a library unit imports type declarations from elsewhere, and wishes to rename them in order to be able to export them without the user having to "with" the units from whence they came. At present, this is only supported by the use of subtype or derived type declarations. The latter requires the use of explicit type conversion all over the place, the former requires renaming of enumeration literals as functions, which are non static and cannot therefore be used as values of discriminants in some circumstances.

IMPORTANCE: IMPORTANT

Ada will fail to become fully accepted as a language for writing modular software.

CURRENT WORKAROUNDS:

None that provide an adequate solution for large programs.

POSSIBLE SOLUTIONS:

Allow renaming of types.

USE OF RENAMES

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 8.5 #5, 9, 14, and 16

PROBLEM:

The programmer should be able to use RENAMES for any construct as a shorthand name for a previously defined object. However, RENAMES should not be allowed as a mechanism to change the number of parameters and defaults--unless this is the means to change the declaration of any object. If so, then renames could be used wherever subtype can be used to override other subtypes. Renames also should not be used to change enumeration types to functions. They should only map to other enumeration names in the designated set to avoid long names. When the programmer provides a constraint that is in conflict with the renames, it should not merely be ignored (another opportunity to miss a sentence in the LRM). For example if a programmer writes the following:

```
procedure x is
        a:integer;
        b:positive renames a;
begin
        b:=-5;  --expect an error!  No, it is an integer
end x;
```

Procedure X is legal. The assignment won't raise an exception or provide the user a warning that Positive is ignored. This problem is further compounded when the renamed item is in a separate compilation unit.

In 8.5 #14, it is too limiting for calls to be only procedure calls and not allowed everywhere within the scope of the renames. The processing of renames should primarily be a symbol table management function.

In 8.5 #15, a subtype is not really the same as a renames. A renames should be usable anywhere the object from the original (sub)type declaration would have been allowed. For the additional level of subtyping, a conversion would have to be provided to allow arithmetic, assignment, comparisons, etc.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Programming standards to limit use to only be used as shorthand for long selector name designators. Programmers certainly must avoid obfuscation in the source code by changing the procedure parameters and defaults. Enumeration types should not be changed to functions.

POSSIBLE SOLUTIONS:

Allow renames for any declared object. Eliminate the numerous special cases where renames can't be used. Do not allow renames to change the form of any renamed object. Delete the analogy to subtypes. If user adds constraints in a renames, then the compilation should not just ignore it but provide a warning and error or take the constraint.

RENAMES FOR TYPES AND SUBTYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 8.5(2)

PROBLEM:

Renaming works for almost every construct in the language except for types and subtypes. For these two, the programmer has to remember that this is a special case and remember that pseudo-renames can be achieved using a subtype:

```
package Foo is

    type Bar is ...

end Foo;

with Foo;
package Blat is

    subtype Bar is Foo.Bar; --Pseudo-renames.

end Blat;
```

This is yet another annoying asymmetry in the standard that does not appear to have any justification.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Allow renaming of types and subtypes. For example, the above code sample could more properly be written:

```
package Foo is

    type Bar is ...

end Foo;
```

```
with Foo;
package Blat is

        type Bar renames Foo.Bar; --Not a new type, not a derived
                type.
        subtype Blat renames Bar;

end Blat;
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

SELECTIVE EXPORT VERSUS RENAMED SUBPROGRAMS

DATE: September 16, 1989

NAME: Kit Lester (on behalf of the Ada-Europe 9X group material supplied by Ivar Walseth)

ADDRESS: Portsmouth Polytechnic
115 Frogmore Lane
Lovedean,
Hampshire PO8 9RD
England

TELEPHONE: +44-705-598943 after 1pm EST
or E-mail to CLESTER @ AJPO.SEI.CMU.EDU
or to LESTERC @ CSOVAX.PORTSMOUTH.AC.UK
or to C_LESTER @ ARE-PN.MOD.UK

ANSI/MIL-STD-1815A REFERENCE: 8.5

PROBLEM:

The standard makes RENAMING_DECLARATION be a BASIC_DECLARATION [3.1(3)], but not also a LATER_DECLARATION [3.9(2)]. In this and other ways, the standard treats renaming declarations of subprograms more like subprogram specifications than bodies, despite the fact that (in a sense) a renaming declaration stands for both a procedure specification and a body.

There are occasions in which it is inconvenient that they cannot be used as bodies, notably when one is trying to better-structure a program by use of selective export and selective re-export. Here is such an occasion with selective export:

```
with TEXT_IO;
package SELECTIVE_EXPORTER is
        type T is private;
        procedure PUT (ITEM:T);
private
        type T is range 1..10;
        package T_IO is new TEXT_IO.INTEGER_IO(T);
        procedure PUT (ITEM:T) renames T_IO.PUT;
                -- Not legal in Ada-83
                -- We don't want to export T-IO.GET etc from
                -- SELECTIVE EXPORTER, nor do we want the
                -- inefficiency of the workaround (see below).
end SELECTIVE_EXPORTER;
```

Apart from scenarios involving generics, this also applies to selective re-export of tapk entries, derived subprograms, and maybe other cases.

IMPORTANCE: IMPORTANT

for certain styles of program structure discipline

CURRENT WORKAROUNDS:

```
package WORKAROUND is
        type T is private;
        procedure PUT (ITEM:T);
private
        type T is range 1..10;
end WORKAROUND;

with TEXT_IO;
package body WORKAROUND is
        package T_IO is new TEXT_IO.INTEGER_IO(T);
        procedure PUT (ITEM:T) is
        begin
                T_IO.PUT(ITEM);
        end PUT;
end WORKAROUND;
```

This has the overhead of one extra call per call of P.PUT (unless one uses pragma INLINE, and that usually introduces compilation dependencies on the body, if the compiler doesn't choose to ignore it).

The issue isn't entirely clear-cut, because there's a counter-argument that the workaround is better anyway:

> From the point of view that it is usually prudent to avoid specs dependant on specs (because it causes obsolescence to cascade), the "with TEXT_IO" on the spec of the original SELECTIVE_EXPORTER isn't too painful, because the spec of TEXT_IO is likely to be recompiled only rarely. But if the selective export or re-export was of something dependent on a spec prone to re-compilation, and we're not trying (or succeeding) to INLINE then the workaround has the advantage that only the body of WORKAROUND (Rather than its spec) is dependent on the prone-to-be-recompiled spec.

This appears to be an inherent weakness of selective re-export, but:

(1) prone-to-be-recompiled specs are usually symptoms of imprudent design, anyway, and the language ought to facilitate prudent design (and leave imprudent designers to stew in their own consequences]; also

(2) experience suggests that re-export tends to be of imports from unusually stable specs

So it seems preferable to give the program designer the choice; only he can assess the relative strength of the approaches in his application.

POSSIBLE SOLUTIONS:

A renaming declaration should be permitted as a body satisfying an earlier subprogram spec.

SPECIFICATION OF PACKAGE STANDARD IN ADA

DATE: October 12, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 8.6, C

PROBLEM:

There is no method of describing package STANDARD in the language.

IMPORTANCE: IMPORTANT

Developers of generic front end compilers will develop their own methods.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Define a set of pragmas like those in 13.7 that declare integer and real types and DURATION.

IMPLICIT CONVERSIONS AND OVERLOADING

DATE: October 31, 1989

NAME: Bill Easton

ADDRESS: Peregrine Systems, Inc.
P.O. Box 192
Bluemont, VA 22012

TELEPHONE: (703) 689-1168
E-mail: easton@stars.reston.unisys.com

ANSI/MIL-STD-1815A REFERENCE: 8.7(13), 4.6(15)

PROBLEM:

The rule for applicability of implicit conversions given in 4.6 (15) is overly complex and confusing.

The rule reads as follows:

> An implicit conversion of a convertible universal operand is applied only if the innermost complete context ... determines a unique ... target type ... and there is no legal interpretation of this context without this conversion.

Apparently, this rule is interpreted differently by different implementers. For example, the following program fragment is interpreted differently by five validated compilers on which I have tried it:

```
function "<" (L,R: INTEGER) return INTEGER;
procedure P(L: BOOLEAN; R: INTEGER);
procedure P(L: INTEGER; R: BOOLEAN);
...
P((1<2)<(3<4), 5<6);
```

The results of the trials are as follows:

> Two compilers reject the above, claiming that the name P in the call statement is ambiguous.
>
> Two compilers accept the above, selecting P(BOOLEAN, INTEGER), with implicit conversion of the literals 5 and 6.
>
> One compiler accepts the above, selecting P(INTEGER, BOOLEAN) with implicit conversion of the literals 1, 2, 3 and 4.

In addition,

> The RM [4.6 (15)] would seem to reject the call statement, as there is one legal interpretation (without the rule) which requires conversion of 1, 2, 3, and 4 and another which requires conversion of 5 and 6; each of these conflicts with "an interpretation ... without this conversion." Thus, there is no legal interpretation of P.

The Implementers' Guide describes a preference rule by which an interpretation with fewer conversions is preferred to one with more conversions, thus selecting P(BOOLEAN, INTEGER).

As near as I can tell, the European Draft Formal Definition appears only to reject call-interpretations which have a non-universal parameter and for which another call-interpretation exists having the same result type but with a universal type as the parameter (function Avoid_Implicit_Conversions). In this example, no interpretations of the "<" functions would be rejected, so both interpretations of P would be legal and thus ambiguous. (Hence, the procedure call is rejected, but for a different reason than that given earlier for rejection by the RM.)

There are four tests in the ACVC which test for the rule in question. Presumably, all five of the compilers tried pass the tests.

A simpler and clearer rule is needed because:

The language should be clearly defined and the same for all implementations.

The current rule in the RM, if implemented, requires testing for conflict with another interpretation for each potential interpretation and each implicit conversion (or tree of implicit conversions under a relational operator) required by it. This seems to require maintaining a list of conversions along with each interpretation during overload resolution.

Clearly the complex cases of this rule are not heavily used, as compiler behavior is not predictable except by experiment.

Finally, the extra complexity will result in extra complexity in any formal definition of the static semantics.

The only difficult case in the interpretation of the rule arises when a relational operator can be either (1) a built-in operator with universal parameters or (2) a user-defined operator returning a type other than BOOLEAN. A user-defined operator returning BOOLEAN should always be rejected in favor of a built-in operator with universal parameters.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Implementers implement something and debug it with the ACVC; implementations differ substantially.

POSSIBLE SOLUTIONS:

Suggested Solution (Follows the European Draft Formal Definition.)

If a function call has an interpretation with parameters of universal type, then this interpretation is preferred over any other interpretation with the same result type.

In any context requiring "any integer type," "any real type," or "any numeric type," an interpretation with a universal type is preferred over any other interpretation.

Note that this solution requires a check at each relational operator. No extra information needs to be

carried up the tree during overload resolution.

The effect of this solution is the same as in Ada 83 except when a relational operator is defined with a result type other than BOOLEAN. (Such definitions are used in at least one proposed Ada binding to SQL.) If any ambiguity is caused by the change, it could always be handled by explicit qualification of the result of the function call with the relational operator, since the function call cannot return a universal type.

TIMER/CLOCK

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 8.8, 9.6, 9.8, and appendix F

PROBLEM:

Many computers do not have a clock and even more applications do not require time information. For embedded systems, there are often many clocks and few have the range, precision, granularity that match the Ada language. Providing timer utilities is not difficult process. Therefore, the timing in Ada should not be required for the compiler implementor to create the runtime package for all the timing routines. This only creates a validation problem and does not impact development. Because of the requirement, Ada is not available for some applications, such as simple process control or robotics. The added flexibility would be useful. New utilities could easily be added to an object oriented design to port the software to a different architecture with different timing. With a user specified accuracy and precision control, then the user does not have to modify or pay for an upgrade to the runtime support package.

IMPORTANCE: IMPORTANT

it is important to cut out the contention for who is managing the hardware resources in an embedded computer. It is also important not to have a preconceived notion as to what time, duration, system'tick means. It is not difficult to overload those definitions when porting programs. It is doubtful that software for a satellite on the way to Jupiter will be ported to an avionics target with different interfaces and targets without having to do something. The overloading of the time meaning should be easy to accomplish within the language design.

CURRENT WORKAROUNDS:

Try to get the compiler vendor to interface to your applications runtime hosted on the machine rather than common runtimes that do not match the hardware

POSSIBLE SOLUTIONS:

Make all the timing syntax/semantics optional. Support more than one package standard. Get rid of time dependent validation testing so that the vendor only has to provide an interface specification for the utilities--you may want to share them anyway.

IMPLEMENTATION WORDING DOES NOT BELONG IN THE LRM

DATE: June 9, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 8.8 #1 and 9 #5

PROBLEM:

In 8.8 #1, "all implementations" is too sweeping a statement. It is untestable and unmanageable. Therefore, such wording should not be part of the LRM. For the parallel processing case, if the results cannot be distinguished by a test case on the language, then the LRM should be silent on how the result was attained. The validation can only test for computation values, states, and attributes, not methods and implementations and internal structure management.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

A major problem in dealing with the LRM.

POSSIBLE SOLUTIONS:

Delete such language as "all implementations" and target processor allocations.

For additional references to Section 8. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 9. TASKS

WHEN AN OBJECT CONTAINS MULTIPLE TASKS

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

When an object (e.g. record type corresponding to a subsystem) contains multiple tasks, rendezvous with one of these tasks involves the user knowing which task does what. Much better for reuse and maintenance (and fits in better with design methods such as HOOD) if rendezvous is with the object, with some declaration in the body or private part of the object to identify which tasks provide which entries.

IMPORTANCE: IMPORTANT

Ada will fail to become fully accepted as a language for writing modular software.

CURRENT WORKAROUNDS:

Various, but tedious for the programmer.

POSSIBLE SOLUTIONS:

Support rendezvous with any object that contains an object or objects of task type(s), with some declaration in the body or private part of the object to identify which tasks provide which entries.

RENDEZVOUS LEADS TO UNACCEPTABLE PERFORMANCE

DATE: August 11, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

Use of Ada rendezvous for inter-task communication or for protection of shared data areas leads to unacceptable performance in some applications compared to more primitive facilities such as semaphores.

IMPORTANCE: IMPORTANT

Developers are tempted to use non-portable, non-Ada constructs.

CURRENT WORKAROUNDS:

Non-portable use of operating system primitives or of Ada run time system internals.

POSSIBLE SOLUTIONS:

Mandate that compilers recognize tasks that are only being used as semaphores, and perform appropriate optimizations.

Define a standard semaphore package.

ADA TASKING

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

In the last 5 years, tomes have been written on the Ada tasking model. It is too complex and has too much overhead for embedded systems to utilize effectively. It also does not fit well with architectures found in embedded systems, e.g., multiprogramming/ distributed processing. The control mechanisms are not responsive to realtime needs. Applications programs are responsible for housekeeping on context switches where the process will not return to the previously selected context. The model does not support the well-known basic scheduling disciplines, e.g., preempt or nonpreempt and resume or nonresume, see Conway's Theory of Scheduling. The problems with tasking efficiency is not the maturity of the compilers, but in the underlying model defined in the language and the validation requirements for the compilers.

IMPORTANCE: IMPORTANT

One of the major goals for the Ada 9x.

CURRENT WORKAROUNDS:

Programming standards to avoid tasking or only initiate a single task and to not use rendezvous of any kind as they are too unpredictable and require too much overhead. Allow the ACVC not to test this section so that the application does not have to absorb a runtime system from a compiler vendor that has little experience with the applications.

Or, write in a language like Modula-2 or object oriented C++ that does not interfere in the target architecture.

POSSIBLE SOLUTIONS:

1. Revamp the tasking model to reduce the overhead functions. This is the single feature that causes embedded systems to not use Ada or prevents the acceptance of Ada as an improvement over conventional methods.

2. Make section 9 optional.

3. Add the 4 scheduling disciplines, [non]preempt/[non]resume, to the task attribute definitions. Also add semantics to allow the tasks (and family tree) to be terminated outside the activation context

in an operating system. This would allow a task controller to start/stop a task (and all the offspring) when a new event occurs and the task will not be resumed but restarted. Abort and terminate do not provide the desired controls--these provide nothing more than an exit with/without an exception raised in the parent.

4. Remove wording for items that can not be tested in the validation--implementation details--e.g., every task has a queue. Allow a "short circuit" for tasks with a single entry/exit point, i.e., at the beginning and end of the task body.

5. Allow only single entry/exits to tasks. (This would greatly simplify the control structures since the compilers would not have to insert duplicate elaboration code at each possible entry point.) The ACCEPTS could only be at the beginning. Then, elaboration code will not be built at every possible entry point. It could be built once and the alternatives treated as a special case statement.

PROPOSAL FOR A (STANDARD) TASK_ID TYPE AND OPERATIONS

DATE: September 29, 1989

NAME: David Emery & Robert Eachus

ADDRESS: MITRE
MS A156
Bedford, MA 01730

TELEPHONE: (617) 271-2815/2614
E-mail: emery@aries.mitre.org
E-mail: eachus@mbunix.mitre.org

ANSI/MIL-STD REFERENCE: 9

PROBLEM:

One of the common extensions to tasking made by many current Ada compilers is a facility to get a system-generated identifier for the current task. It is often necessary to obtain the identity of a task, and to pass it around as a parameter to other code, which can make a decision based on task identity.

For instance, consider a Lock package:

```
package Locks is

    cant_lock : exception;

    type lock_type is (read_lock, write_lock);

    procedure lock (l_type : lock_type);

    procedure wait_to_lock (l_type : lock_type; wait_time :    duration);

    procedure unlock (l_type : lock_type);

    end Locks;
```

With the following intended semantics:

Many tasks can obtain a read_lock, but only one task can obtain a write_lock. A write_lock precludes all read_locks.

Lock works as follows:

1. If there is no active lock, the requested lock is granted to the calling task.

2. If the caller requests a read_lock, and only read_locks have been granted to other tasks, the lock is granted.

3. If the caller requests either a read_lock or a write_lock, and the current task already holds the appropriate lock, the request is granted.

4. If the caller requests a read_lock, and the current task holds a write_lock, the request is granted.

5. Otherwise, cant_lock is raised;

Wait_to_lock works the same way, but the procedure "blocks" until the requested lock can be granted, or the wait_time runs out.

Unlock releases the lock held by the current task.

The problem is implementing this package reliably requires two facilities: First, each task must have an unique ID. This is used to associate a lock with a task. Second, there must be a way for an arbitrary procedure to obtain the identity of the task 'enclosing' it.

This request was discussed at the recent ACM SIGAda Ada Language Issues Working Group (ALIWG) meeting at Tri-Ada. In addition to the issue described above, the working group believes that it is important to be able to reliably determine the identity of the task calling the "server task" in a rendezvous.

ALIWG discussed whether any ordering should be imposed on task_id's. This would make it easier to implement data structures based on task_id's. More specifically, it would be nice to be able to convert a task_id into an integer, permitting both comparison operations and numeric based operations, such as hashing, for efficient implementation of data structures indexed by task_id. There was no clear consensus for this, although no one argued against the principle of providing facilities to make it easier to build task_id data structures.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The common approach is to provide each task with its identity via an entry call. As has been documented in the literature, this causes a tremendous synchronization bottleneck for the 'master' to give each task its ID. Then this task ID is made accessible via programming conventions and tricks, that permit a procedure to call a function returning this ID. Sometimes, getting these tricks to work correctly requires passing the task ID all over the place as an extra parameter to procedure calls. In any event, such techniques are subject to programmer mistakes, and are clearly inefficient.

This feature is particularly important for reusable software components (such as the lock example_ that need to make decisions based on their caller's enclosing task.

Although the calling task could pass its ID as a parameter to the server task, there is no guarantee that the task ID passed actually represents the ID of the caller (i.e. the caller could "lie" and pass the identity of some other task.)

POSSIBLE SOLUTIONS:

The basic idea is to provide a private type in package System, and a function that returns a value of this type, representing the task ID:

```
package System is
   ...
```

```
        type task_id is private;

        function current_task_id
                return task_id;

        function caller_of
                return task_id:
        function image_of (t_id : task_id)
                return string;

        ...
private
        typetask_id is <implementation defined>;
end System;
```

A Task_id should be guaranteed to be unique throughout the lifetime of the program. One technique for implementing guaranteed unique task_id is to use a timestamp as part of the task_id.

Task_id'constrained should always return true.

The operations on task_id are those of any private type:
assignment
membership
equality
plus two specific operations, one that returns the task_id of the current task, and the other that returns a string representation of the task_id. When called from a library unit, current_task_id should return the task_id of the "environment task". There is no way to convert from a string (back) to a task_id.

The cost on existing implementations should be minimal, since this information can be stored in the task control block, and initialized when the task is created. This implementation of current_task_id is trivial; it obtains this field stored in the current task's control block. Note that when in a rendezvous, this function must return the task_id for the accepting task (not the calling task) (even in the presence of the Habermann-Nassi optimization and other techniques . . .)

The image_of function, although not necessary, provides a very useful debugging tool. Note that there is no corresponding value_of function. (Note that if the language permitted user-defined attributes, this function wouldn't be necessary. Instead, there could be an implementation of the 'image attribute for the type task_id.)

It might be desirable to add a function that returns the task_id of the "environment task". In particular, components could use this to determine when they are called by a task other than the "environment task", and modify their behavior to be correct in the face of multiple tasks.

In addition to the function returning the current_task_id, there should be another function, caller_id, also returning task_id. This function is valid when called 'inside' of a rendezvous (either directly or indirectly as part of the accept block). It will return the task_id of the calling task. It should raise tasking_error when it is called and not in a rendezvous.

ASKING FOR A MORE PRECISE DEFINITION OF THE SYNCHRONIZATION POINTS OF A TASK

DATE: August 8, 1989

NAME: Patrick de Bondeli

ADDRESS: CR2A, 19 Avenue Dubonnet
92411 - COURBEVOIE - CEDEX, FRANCE

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

The present definition of the synchronization points of a task is incomplete and imprecise; it should be clarified. The situation can be summarized as follows:

a. LRM 9 (2) provides a general definition of synchronization points;

b. Specific synchronization points are defined in different sections of LRM 9;

c. A number of specific execution points of a task are synchronization points according to LRM 9 (2), but are not explicitly defined as such in any section of LRM 9.

This situation sometimes results in an absolute contradiction:

> LRM 9.5 (10) precisely defines the location of the synchronization points of the beginning of a rendezvous after the evaluation of the actual parameters of the call on the caller side and after the evaluation of the entry index (if any on the accepter side;
>
> LRM 9.10 (6) contradicts LRM 9.5 (10) by stating that the synchronization point on the accepter side is at the start of the accept statement (that is before the evaluation of the entry index).

Of course the definition of LRM 9.5 (10) should be preferred to its contradictory statement in LRM 9.10 (6) since it is more natural and compliant with LRM 9 (2).

An Analysis of Ada Tasking, which was written in conjunction with the production of the formal definition of Ada sponsored by European Commission, enumerates the occurrence of case c where a given point of execution should be a synchronization point according to LRM 9(2) (i.e., a point where a task cannot proceed independently because there is an interaction with the outside world) but is not explicitly defined as such; they are the following

> Calling a Subprogram or Activating a Task - There is an interaction with the outside world at the points where the task starts and ends verifying that the body of the called or activated object has been elaborated. So when a task T activates another task T', T has several synchronization points: starting and ending verifying that the body of T' is elaborated, starting and ending the activation of T'.

Completing the execution of a frame - This is a potential interaction with the outside world, since the task, at the end of this action, must wait until all tasks depending on this frame are terminated.

Terminating - This is a potential interaction with the outside world since the termination must be flagged to the parent task.

Evaluating the attributes TERMINATED or CALLABLE -
Giovini and Zucca (1986) consider them as synchronization points, but we do not agree on behalf of the fact that the interaction with the outside world does not imply here any synchronization in the sense of LRM 9 (2) (both the "evaluating" task and the "owner" task keep proceeding independently here).

Calling an IO operation - This point is not mentioned by Giovini and Zucca (1986); we however believe some IO operations must involve here a synchronization point for the calling task T in order to suspend T until the invoked IO operation can be completed.

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

REFERENCES/SUPPORTING MATERIAL:

Givoni, Allesandro and Zucca, Elena. An Analysis of Ada Tasking
December 1986. *Companion paper of the Ada Formal Definition* Sponsored by the European Commission.

De Bondeli, Patrick. Session Summary - Asynchronous Transfer of Control. Proceedings of the International Workshop on Real-Time Ada Issues - Moretonhampstead, Devon, UK. June 1988. Also published in Ada Letters (Special Edition) Vol. VIII N.7 Fall 1988

PROTECTION AGAINST VIRUSES AND TROJAN HORSES

DATE: October 20, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@SDEVAX.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 9, 13

PROBLEM:

Protection against viruses and trojan horses is becoming a more serious issue in programs. This is especially a problem with Ada since it involves large programs with portions that may be derived from a variety of sources, not all of which are equally trusted. Pragmas and procedures should therefore be added to Ada which can be used to increase the security of the system given that some of the code may not be trusted.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

1. Include a pragma that would indicate that memory management hardware, where available, should be used to insure that memory related to a package can't be affected by code in other packages.

2. Provide a pragma that will allow a "resource name" to be assigned to a package. The run time system would then verify that no other package could be assigned the same resource name.

3. Provide a pragma for procedures and functions that would restrict the packages that could call those procedures to those whose packages have specific resource names.

4. Provide a function that can be inserted in procedures and functions to determine the resource name of the package making the call.

5. As part of the implementation, provide a means for the operating system to determine the resource name for the package making system calls. This could then be merged with security software to insure that accesses to resources are made from trusted portions of the program. For example, requests to access printers or communications channels could be restricted to a package with a specific resource name.

It is not clear that all of these steps are feasible with existing technology. However, I still feel that measures of this type should be considered. Even though implementation of these pragmas may require extensions of the operating system, especially if a secure environment is desired, it would be advantageous to have the format specified in the language description so that it would be automatically implemented when

placed on a secure system.

RENDEZVOUS BETWEEN INDEPENDENT PROGRAMS

DATE: October 20, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@SDEVAX.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 9, 13

PROBLEM:

In some cases, it may not be practical to write an entire system as a single program. For these cases, it should be possible to have rendezvous between independent programs as well as between parts of the same program By including a description of the format as part of the language description, it will be possible to make the systems more portable.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use of system dependent code

POSSIBLE SOLUTIONS:

There are a number of possible approaches.

1. Use an electronic mail system to talk between different programs. The programs would have the ability to give themselves a specific user name and would then send and receive mail from the other processes. (Both the current X.400 mail system and the means used by IBM VM/CMS to exchange files between users should be examined.)

2. Provide a means of naming communication lines. Connection between the programs would then be established when two programs hook onto the same line.

Given procedures of these types, the system should then be able to have a task that will wait for the arrival of messages or electronic mail. One possible way would be to have the system define the arrival of mail as an interrupt activity and then use the existing procedures for handling interrupts to trigger activities.

SIMPLE PARALLEL THREADS

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

The user has no tools to indicate that parts of his program are independent and can (or should, to control efficient execution) be executed in parallel where the architecture allows such, unless by employing the task concept which is too burdensome, verbose, and inappropriate in this case.

For example, for simple independencies like in

```
parallel
        do => BLOCK_A;
        do => BLOCK_B;
end parallel;
or
parallel for I in A_RANGE loop
        ITERATION;  -- like:  U(I) := V(I) + W(I);
end loop;
```

the user cannot easily indicate :

- that parallel facilities should be exploited,
- that the different parallel threads obtain different physical processors,
- and that every ITERATION in the example simply knows the unique (iteration) value of the loop parameter and the unique identity of the processor it is employing (in the above example with the statement U(I) := V(I) + W(I); , the passing of the values of I, such that every branch knows which components to take, requires an unacceptable amount of overhead in programming entry calls and accept statements).

Employing tasks here would serialize the different threads and thus spoil all effects of possible parallel speedup.

What we require is:

a. Ada semantics to allow parallel execution of apparently independent program parts that do not require communication between the branches,

b. a simple way of giving initial values to tasks that are being activated.

IMPORTANCE: IMPORTANT

This requirement, though not stated to be essential in the sense of the word 'ESSENTIAL' as defined in

the format, is actually ESSENTIAL for many implementors of Numerically Intensive Computing (NIC) methods. In other words, the revised standard is unlikely to be accepted by the community making the suggestion if this revision request is not satisfied.

CURRENT WORKAROUNDS:

Hardly present.

POSSIBLE SOLUTIONS:

More freedom allowed by the Ada semantics for actions that are currently considered to be sequential, if necessary only upon explicit request for paralleling some independent parts (e.g. by a pragma).

SUPPORT PRAGMA SHARED ON COMPOSITE OBJECTS

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

Currently support for shared variables is inadequate in Ada, as it applies only to objects which can be loaded/stored in a single non-interruptible instruction.

However, many systems are best designed by using shared memory for communication. Ada makes this difficult by requiring explicit task synchronization for all use of shared memory (other than that allowed by pragma SHARED). Furthermore, if the application uses a "lock"/"unlock" discipline using a central task to manage the lock but not to actually perform the operation, then it is difficult to ensure that the object will be unlocked, and to efficiently compose subprograms each of which needs exclusive access.

IMPORTANCE: ESSENTIAL

Shared memory is an important programming paradigm which is not adequately supported by Ada. If this is not fixed, programmers will continue to use nonportable, unsafe, and/or inefficient mechanisms to use shared memory.

CURRENT WORKAROUNDS:

The best workaround is to put all operations on shared memory into accept bodies of a single task. However, this introduces overhead in the cases where an object is known to be unshared, or to already be held exclusively (perhaps because it is a component of an exclusively held object).

POSSIBLE SOLUTIONS:

A possible solution is to support pragma SHARED (or equivalent) on limited composite types, with the semantics that the declaration is associated with a *type*, and any subprogram taking that type automatically gains "stable" access to the object, if it is not already held by the caller. "Stable" access means that IN parameters are locked for (shared) read-only access, and IN-OUT and OUT parameters are locked for exclusive write access. The lock is automatically released on exit from
the subprogram (whether normally or by exception).

Calls on such a subprogram are delayed if the locks cannot all be set. Conditional/timed entry call syntax

may be used to call such subprograms to avoid indefinite delays. No locks are set until all locks can be set.

Any type declared SHARED is required to be limited, and the semantics for functions returning a shared type are defined analogous to task type, namely that the same shared entity is designated by the returned object as that designated by the return expression (i.e. no implicit copying is performed).

Note that there are no special controls on access-to-shared type, which allows an access-value referring to a shared object to be passed between subprograms without any locks. However, when the access value is dereferenced as part of passing the designated object to a subprogram, locks will be set.

Note that given this shared-type mechanism, task types which are actually monitors can be simulated as access-to-shared-types, with entry calls being calls on subprograms which take the designated (shared) task object as an implicit IN OUT parameter.

To fully simulate task types, it would also be necessary to be able to spawn a subprogram call as an independent thread of control, passing it a shared object to allow communication with it. The spawned subprogram call would begin as soon as all required locks could be set. Furthermore, it would be necessary to allow boolean "guards" on subprograms and to allow timed or conditional spawning of subprograms.

LOOSELY-COUPLED INTER-TASK COMMUNICATION

DATE: October 16, 1989

NAME: Samuel Mize

DISCLAIMER:

The views expressed are those of the author, and do not necessarily represent those of Rockwell International.

ADDRESS: Rockwell/CDC,
3200 E. Renner Road M/S 460-220,
Richardson, TX 75081

TELEPHONE: (214) 705-1941

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

For this discussion, "tightly-coupled" tasks are those which need to share a rendezvous (e.g., which need to synchronize their activities or share computations). "Loosely-coupled" tasks are those tasks which only need to establish a flow of data between them (e.g., using an input buffer, or the classic producers-consumers problem).

Ada provides native facilities only for tightly-coupled tasks. Two loosely-coupled tasks must be implemented as three tightly-coupled tasks (the two "actual" tasks and another to manage communications).

These extra tasks can, in some cases, create some task-switching overhead. They can also confuse a reader of the code, increasing maintenance costs. If hidden in a package body, they can create timing or deadlock problems which will be hard for the package user to understand and debug (impossible, if the body is not available for inspection).

IMPORTANCE: IMPORTANT

Providing a loosely-coupled task communication construct would help programers write code faster. Maintenance programmers would understand these programs more [illegible] and accurately. An entire class of programming errors (mistakes in writing buffer tasks) would be eliminated.

CURRENT WORKAROUNDS:

1. Use a buffer task between the two loosely-coupled tasks. This task always accepts entry calls from the consumer whenever the buffer is not empty. (or, it can always accept an entry call from the consumer, and return either an item from the buffer or a flag indication that the buffer was empty.)

2. Code per work around 1, but hide the synchronization task inside a package implementing the buffer.

3. Implement a semaphore using a SHARED variable. Use this to control access to the buffer.

Preferably, access to the buffer is encapsulated in procedures that properly manage the semaphore.

All of these are prone to programmer error. The first is fairly clean in Ada, but raises questions of task-switching overhead if there are a number of data flows being managed in this way. Workaround 2 creates the cleanest, most readable code (unless the hidden task creates a deadlock or timing problem, and the maintenance programmer doesn't even know it is there).

POSSIBLE SOLUTIONS:

Provide a primitive for loosely-coupled inter-task communications. This is a specific instance of providing packages with mutual exclusion properties, which I have addressed in another revision request.

One design for loosely-coupled inter-task communications would be a pair of standard generic packages, Synchronized_Queue and Asynchronous_Queue, to be instantiated for each data flow between loosely_coupled tasks. (The names are inconsistent to keep a single letter error from compiling). These packages are specified below. They can be implemented as packages that contain synchronizing tasks. The reasons for making them standard packages, like Text_IO, are 1) to encourage more readable application code, and 2) to let the language implementers provide the same types of coordination more efficiently than an end-programmer can with buried tasks, if their compiler architecture will allow it.

In both packages, Maximum_Queue_Size = 0 implies an unbounded buffer, presumably using linked lists. Any list "buckets" are released with unchecked_deallocation; it is implementation-dependent whether this actually reclaims any memory. A value of Maximum_Queue_Size greater than zero implies some form of compile-time memory allocation for the buffer; for example, an array implementing a circular buffer. No run-time allocation or deallocation occurs in this case.

```
generic
        type Item_Type is private
        Maximum_Queue_Size : in Natural := 0;
package Synchronized_Queue is

        procedure Add_To_ Queue (Item : in Item_Type);
        -- If (Maximum_Queue_Size > 0) and (number of items in queue is equal to --
        Maximum_Queue_Size), calling task hangs up as if at a hidden entry call until an item is removed
        -- from the queue.
        -- Adds a copy of Item to the synchronized queue.

        procedure Get_From_Queue (Item : out Item_Type);
        -- If queue is empty, calling task hangs up as if a
        -- hidden entry call until something is put into the
        -- queue.
        -- Removes the next item from the synchronized queue and     --     returns it.

end Synchronized_Queue;
```

```
generic
        type Item_Type is private
        Maximum_Queue_Size : in Natural := 0;
package Asynchronous_Queue is
```

```
        procedure Add_To_Queue (Item : in Item_Type;
                                       Item_Added : out boolean);
        -- If (Maximum_Queue_Size > 0) and (number of items in
        --       queue is equal to Maximum_Queue_Size), returns with
        --       Item_Added = false.
        -- Else, adds a copy of Item to the asynchronous queue
        --       and returns with Item_Added = true.

        procedure Get_From_Queue (Item : out Item_Type;
                                       Item_Found : out boolean);
        -- If queue is empty, returns with Item_Found = false,
        --       Item is not defined.
        -- Else, returns with Item_Found = true, Item = the next
        --       item from the asynchronous queue.

end Asynchronous_Queue;
```

'SIZE ATTRIBUTE FOR TASKS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
FAX: +46 758 15133

ANSI/MIL-STD-1815A REFERENCE: 9, 13.7.2

PROBLEM:

It is not possible to set stack size for a task, but only for a task type. This means that 1) it is impossible to tailor the stack size for the various instances of the task type, and 2) if only one task is actually used, it is necessary to declare a task type anyway. This is (mildly) confusing to the untrained eye.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Extend the applicability of the 'SIZE attribute.

EXCEPTIONS RAISED ON OTHER TASKS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 9, 11

PROBLEM:

The only way to abort a task is through the abort statement. Firstly, this guarantees nothing (since the definition is in terms of synchronization points, which the task may very well never reach), secondly, it does not allow the aborted task to perform any cleanup actions.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Make it possible to do

```
E: exception;
task T is ... end T;

...
raise E in T;
```

which causes E to occur in T.

ASYNCHRONOUS QUEUES

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

We tend to build out real time systems around the concept of asynchronous queues, rather than around rendezvous. These queues are mostly used for program-to-program communication, and is therefore - unfortunately - not in the scope of the LRM. However, it seems to me that it would be easy to add the queue paradigm to the Ada vocabulary (see Solution below).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Only ugly and/or inefficient alternatives (such as for instance building buffer tasks).

POSSIBLE SOLUTIONS:

```
task T is
        entry E1(...);
        queue Q1(...);
end T;
```

where the restriction for queue declarations is that it may only have in parameters (cf. functions as opposed to procedures!) The accept statement has the obvious syntax; the semantics of calling a queue is to deposit a message and return, if there is space in the queue.

PROBLEMS WITH I/O IN MULTITASKING APPLICATIONS

DATE: October 20, 1989

NAME: Gene K. Ouye

ADDRESS: Home: 6016 Mardale Ln.
Burke, VA 22015

Work: Attn: Gene K. Ouye, Mailstop TM305
Planning Research Corporation
1500 Planning Research Drive
McLean, VA 22101

TELEPHONE: Home: (703) 451-3267
Work: (703) 556-1838

ANSI/MIL-STD-1815A REFERENCE: 9, 13, 14

PROBLEM:

Due to the synchronous nature of I/O in Ada as it is specified in the LRM, when a task issues an I/O request (or calls a procedure which results in the issuing of an I/O request) it suspends execution until the I/O completes. Logically, this is what we would expect. That is, all of the data transfer routines in the predefined I/O packages will not return until data has been received or transmitted (even if it is only to or from a block buffer, i.e., no physical I/O has to occur).

This seems to mesh well with Ada tasking, which uses a synchronous model. For example, inter-task communication takes place via the rendezvous, a synchronous mechanism (unlike message passing, which can be performed asynchronously) which functions like a Remote Procedure Call. In a true multitasking application, synchronous tasks and synchronous I/Os should work well together. For each device (logical or physical) on which I/O is to occur, a handler task can be defined which will manage all of the I/O to and from the device. For example, in a teleconferencing application, for each user who logs in to the application, a terminal handling task can be allocated which will be dedicated to serving the user at that terminal until he logs out. These terminal handling tasks will spend most of their time hung on a read to the user terminal, waiting for a key depression.

Intuitively, one would expect that when the handler for User A issues its read to the terminal, it would be suspended until User A performed some action. Meanwhile, the handler for User B would now be eligible to run. It could also issue a read to its corresponding terminal which would suspend it until User B responded. At this point, both of the handler tasks have an I/O outstanding, and if either user enters something at this terminal, the outstanding read will complete for his handler task which should then become eligible for execution.

However, in the current definition of the language, tasking and I/O are not required to be interrelated. Therefore, an Ada compiler vendor can provide a validated compiler (and associated run-time system) which will produce a program in which an I/O issued from one task will block the execution of all other tasks in an Ada multitasking application until that I/O completes. In User A issues its read, the entire

teleconferencing application is suspended until the read completes (when User A presses a key). Thus, the handler for User B cannot even issue its read until User A responds.

This problem can occur because the run-time system included as part of each executable Ada application is not required to recognize an I/O as a significant event in a multitasking application. A task waiting for an I/O to complete (or for any external event to occur, for that matter) can still be considered "running," because it has not reached a point considered as significant by the run-time system (e.g., an accept statement, an entry call, a delay statement, access to a shared variable, etc., all of which the LRM requires to be significant). In other words, because the task was not suspended by the Ada run-time system, it is assumed to be executing.

This is entirely counter intuitive and can result in attempts to implement programs which are logically correct, and which intuitively should perform acceptable, but which instead are dogs whose performance is never acceptable with more than one user. In fact, with many validated Ada compilers, it is impossible to create a multitasking application which will perform acceptably. With most of those compilers, it is necessary to write each task as a separate process (i.e., a separate Ada program) under the operating system, and then make use of whatever tasking facilities are provided by the operating system and ignore the tasking provided by Ada.

IMPORTANCE: ESSENTIAL

If I/Os are not considered significant tasking events by Ada compilers/run-time systems, then multitasking applications which requires multiple simultaneous I/Os cannot be written in Ada. In addition, because Ada multitasking applications which run in one environment may not run in another environment, this issue affects portability. It is very difficult to write portable code when interaction between two such basic features of the language varies so greatly between environments.

CURRENT WORKAROUNDS:

There are two workarounds which I have used, both of which require asynchronous I/Os to be supported in some manner by the run-time system and the target operating system. In the simplest workaround, an I/O package can be written whose body contains calls to the asynchronous I/O routines. Immediately after the asynchronous call is issued, the I/O routine enters a loop which contains a delay statement followed by a test of the I/O status. If the I/O has completed, then the routine exits the loop, otherwise it loops around to delay again. The problem with that solution is that every task which issues an I/O via that package will be executing in a polling loop, thus taking away CPU time from other tasks which have legitimately required it. Additionally, each time the task goes through the polling loop, at least two unnecessary task switches will have occurred: one to start the task so that it can check the I/O status, and a second when the task delays again because the I/O still has not completed.

One way to reduce that thrashing is to create in the package body an I/O task which processes each I/O request. The task will have an entry point for new I/O requests to be queued, and an array of entry points, one corresponding to each potential issuer of an I/O, which signal an I/O completion and return the results. The procedures which are visible in the specification of the I/O package will do two things: first, they will call the entry to queue I/O requests, second, they will immediately call their assigned I/O completion entry. The hidden I/O task will take each incoming I/O request and issue the appropriate OS call which will perform the I/O asynchronously. As the I/Os complete, the I/O task will accept the I/O completion entry call which corresponds to the issued I/O. In this way, the terminal handling task which made the call to the procedure in the I/O package will be suspended by the Ada run-time system and will not execute again until its I/O request has completed. Only one tasking the system will be in a polling loop, and that task will poll for all tasks which have issued I/Os through the package. This method also has the added benefit

of being able to terminate tasks which are waiting for I/O completions.

In environments which allow mapping of task entries to asynchronous I/O completion routines, the one I/O task in the package body can be written as two tasks, one to queue I/O requests (and issue them), the other to handle the I/O completions. This workaround contains no unnecessary polling.

In Ada environments in which there is no interface to an operating system facility to issue an I/O asynchronously, there is no workaround. If the target operating system treats the multitasking Ada program as one process, then if any task issues an I/O, the whole application has issued the I/O. If the only available I/O routines are synchronous, then the entire application is suspended until the I/O completes.

POSSIBLE SOLUTIONS:

There should be some way of integrating I/O and tasking into the Ada runtime system so that this problem does not occur. As a minimum, the I/O packages which are currently defined (Direct_IO, Sequential_IO, etc.) should be enhanced or new packages should be defined to incorporate either an asynchronous return (perhaps to a task similar to an interrupt handler or to allow checking of an I/O status for completion. In some environments, this would be a major undertaking, because the target operating system does not allow an application to issue multiple asynchronous I/Os. However, in target environments which do not allow them, support for asynchronous I/O by should be required, and use of them by individual tasks in a multitasking application should not adversely affect the operation of the entire application.

A PEARL-BASED APPROACH TO MULTIPROCESSOR ADA

DATE: October 5, 1989

NAME: Karlotto Mangold

ADDRESS: AEG - ATM Computer GmbH
ATM V1
Postbox 2328
7750 Konstanz
Federal Republic Germany

TELEPHONE: +49 7531 807-235

ANSI/MIL-STD-1815A REFERENCE: 9, 13

PROBLEM:

Embedded systems are often implemented on distributed systems. These probably heterogenous systems should be programmable in Ada. Till today the philosophy of an Ada program enforces a totally transparent implementation which leads into inefficiencies. There should be at least a possibility to take care of the system structure during programming. Another must is the ability to define reconfigurations for emergency cases.

IMPORTANCE: ESSENTIAL

For implementing embedded systems on distributed hardware structures in a machine independent way.

CURRENT WORKAROUNDS:

Omit Ada tasking and Rendez-Vous, using the powerful package My-NET-OPERATING-SYSTEM which offers machine dependent solutions, influencing even the algorithms.

POSSIBLE SOLUTIONS:

Take a PEARL-Like approach

ASYNCHRONOUS MESSAGE PASSING CAPABILITIES LACKING IN PARALLEL PROCESSING

DATE: October 19, 1989

NAME: Philippe Collard

ADDRESS: California Space Institute
A-021, UCSD
La Jolla, CA 92093

TELEPHONE: (619) 534 6369

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

The use of parallel processing is particularly fitted for some applications. Thus, writing these applications in Ada makes a lot of sense since parallel processing constructs are integral part of the Ada language. However, the lack of asynchronous message passing capabilities may cause the design of these applications with Ada to be very inefficient.

For instance, let us suppose a classification algorithm where the computation of classification parameters is handled by a separated task for each possible class. The overall classification process is controlled by a parent task that ensures proper synchronization and gathers classification results. These sort of processes are generally iterative, thus extremely time and resources consuming. In some cases, one of the tasks computing classification parameters will detect very early that the set up for the current iteration will not produce an improvement of the results. Thus the current iteration should be stopped and new parameters should be computed for a new iteration. Unfortunately, it is impossible for the control task to asynchronously send an message to the other computation tasks instructing them to stop computation and get ready for a new iteration. This is due to the purely synchronous nature of Ada task-to-task communication constructs.

One could design the computation tasks such that they "poll" the control task at regular interval to check if they should continue working on the current iteration. Unfortunately, this solution will lead to a great waste of resources (due to the extra communication needed, ie.e., extra context switches). Additionally, this solution does not allow for a full utilization of parallel architectures since the necessity to "poll" the control task serializes the computation process by creating a critical section.

What is needed is a way for a parent task to asynchronously "interrupt" its children in order to engage in communication activities. We realize that this is somewhat contradictory to the philosophy of Ada tasking constructs but we believe that such capabilities should be considered for inclusion in the revised standard to allow for a full utilization of the Ada language for programming parallel architectures.

IMPORTANCE: IMPORTANT

We feel that this request is <u>important</u>. However, the impact on implementations would be significant.

CURRENT WORKAROUNDS:

See problem statement.

POSSIBLE SOLUTIONS:

We propose the introduction of a new class of objects called EVENTS. Events are declared like exceptions. Example:

```
Abort_Current_Iteration: event;
```

Events can only be declared in tasks. Events are "activated" with a RAISE statement like in:

```
raise Abort_Current_Iteration;
```

Events can only be raised by tasks.

Events are handled through EVENT HANDLERS. Event handlers are declared at the end of a block BEFORE any exception handlers. Example:

```
event:
when Abort_Current_Iteration=>
        ...
exception
    ...
end My_Task;
```

When an event is raised, any task that contains an event handler for this particular event is interrupted and execution proceeds with the event handler. Any statement, except GO TO, can appear in event handlers, including entry calls and accept statements. When the sequence of statements of an event handler has been completed, execution resumes in the interrupted tasks where it was suspended prior to the raising of the event unless one of the statements raises an exception. In that case, control is passed to the exception handler if one is provided. The scoping and visibility rules provided for exceptions also apply to events.

In the case of the classification process we described earlier, a computation task could be structured as follows:

```
loop
    COMPUTE:declare
    begin
        accept Start;   - Wait for control task to send go ahead
        ...
    event
        when Abort_Current_Iteration=>
            raise Stop_This_Iteration;

    exception
        When Stop_This_Iteration =>
            null;

    end COMPUTE;
```

```
end loop;
```

STANDARD REAL-TIME LIBRARY

DATE: October 1, 1989

NAME: Pierre A. Parayre

ADDRESS: Direction de l'Electronique et de l'informatique
14.Rue St-Dominique
75997 Paris-Armees
France

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

Ada provides only one synchronous primitive: the rendezvous. That primitive is not sufficient for the practical implementation of highly constrained real-time systems including distributed systems.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Many implementations use non-Ada executives (ARTX.PSOS., etc...)

POSSIBLE SOLUTIONS:

Provide the definition of a standard real-time library which defines asynchronous primitives. The semantics of such primitives can be defined in terms of rendez-vous. They can also be directly implemented in the real-time executive.

The example of such a library called EXTRA is joined including:

-- the Ada interfaces
-- the semantics in terms of rendez-vous

Some minor changes in the Ada LRM would facilitate the introduction of such a library.

ASYNCHRONOUS INTERRUPTS IN ADA

DATE: June 6, 1989

NAME: Jeffrey S. Ayers

ADDRESS: GTE Electronic Defense
P.O. Box 7188
100 Ferguson Drive
Mountain View, CA 94039
Bldg 7 Mailstop #7G35

TELEPHONE: (415) 694-1645

ANSI/MIL-STD-1815A REFERENCE: 9

PROBLEM:

There is no effective method in Ada for a task to interrupt another task, rendezvous with that task, and communicate with that task. The interrupted task should be allowed to exchange information with the calling task and either terminate itself or resume its execution from the location of the interrupt.

IMPORTANCE: ESSENTIAL

This revision is essential since Ada was built to implement embedded systems. These systems must be flexible and resilient to benign interrupts such as user requests for data.

CURRENT WORKAROUNDS:

In an effort to notify a task of an asynchronous event, some applications use polling, so a task can perform some important operation in a timely fashion. This mechanism can be accomplished through shared variables. This method violates the rules of modularity by tightly coupling modules. Plus, if the asynchronous event must be done quickly, then there is a lot of overhead involved since the shared variable must be polled often.

In an effort to avoid Ada's abort statement because of its unpredictable nature, other applications reset stack pointers to remove the task's call frame off a process's stack. However, this does not allow the task to communicate directly with the aborting task nor does it allow the aborting task an opportunity to do some cleanup work. In addition, this forces low level details and machine dependence into an application which is the very thing most applications want to avoid unless their dealing with hardware directly.

Currently, Ada provides a mechanism called "interrupt Entries". These entries attaches a task to a hardware location in memory using address clauses. The general use of this construct is for hardware interrupts. However, software can call these entries to simulate interrupts. Unfortunately, some implementations do not support address clauses and/or interrupt vectors which are used to support Interrupt Entries. Thus, applications could use this construct if the underlining implementation supports it. If an application did use interrupt entries, there would be a serious question of portability. Especially, if the application was targeted for more than one type of machine. Most likely, the application would be forced to use one of the other solutions.

POSSIBLE SOLUTIONS:

In an effort to find a solution, some have suggested allowing tasks to raise exceptions in other tasks. This would allow a task to execute an exception handler and perform cleanup work before terminating. However, this form of interrupt does not allow communication between the task that raised the exception and the task that will terminate. In addition, the only benefit of this form of interrupt would be the graceful termination of a task. But, the task may not wish to terminate its execution.

This illustrates a need in Ada for an effective method for a task to interrupt another task without a major impact to the Ada language. I propose a straight forward change to Ada by changing the INTERRUPT ENTRY to the following form:

```
entry [interrupt] <ENTRY_NAME>[(parameters)];
```

This interrupt entry provides a means to allow asynchronous communication between tasks. There can be one or more of these interrupt entries per task. This would be the only syntactical change to Ada. The semantics of this entry is that a task can be interrupted as long as it is not rendezvousing with another task. In which case, the calling task would wait until the rendezvous was over. The current state of the called task must be saved, and the code associated with the interrupt entry would be executed. The code could terminate the task, or the task could resume where it left off when it completed executing the interrupt entry. The following is example of an interrupt entry.

```
task T is
        entry interrupt STOP_TASK (P1:  out INTEGER);
        entry           START_TASK(M1: in   INTEGER);
end T;

task body T is

        T1:  INTEGER ;

        begin

                loop
                        select
                                accept STOP_TASK (P1: out INTEGER)is
                                        P1 := T1 ;
                                end STOP_TASK ;
                                -- Execute shutdown statements
                                exit ; -- Terminate the task
                        or
                                accept START_TASK (M1: in INTEGER)is
                                        T1 := M1 ;
                                end START_TASK ;
                                -- Execute some number of statements
                        or
                                delay 10.0 * HOURS ;
                                -- Execute some number of statements
                        end select ;
                end loop ;
        end T ;
```

Task T provides two entries: one that begins some processing and one that can be used to interrupt that processing. The body provides two accept statements that are executed when a rendezvous takes place with another task. Additionally, the code after the accept block STOP_TASK will terminate this task. The following is an example of a task calling the STOP_TASK interrupt entry:

```
task S ;
task body S is

        X:      INTEGER ;

        begin

                -- Execute some number of statements
                T.STOP_TASK( X ) ;
                -- Execute some number of statements

        end S ;
```

The call to interrupt entry STOP_TASK causes the current state of task T to be saved. Then task S and task T rendezvous, and the object x receives the value of object T1. Both tasks then proceed to execute their next statements. When task T reaches the EXIT statement, it will terminate. On the other hand, if we remove the EXIT statement, task T will then resume its execution at the statement prior to the interrupt. If task T was at the delay statement, then the call of interrupt entry STOP_TASK would not cause the current state to be saved since the task; was already waiting for a call to an entry.

This form of interrupt entry provides another layer of abstraction from the underlining implementation of the interrupt. Thus, each implementation could determine how best to provide the mechanism for its machine. In addition, the interrupt entry would allow asynchronous rendezvous and communication between tasks. Tasks could exchange information and then resume their execution or terminate.

REQUIREMENT FOR INTER-TASK COMMUNICATIONS IN A DISTRIBUTED/PARALLEL/MULTI-PROCESSOR ENVIRONMENT

DATE: August 25, 1989

NAME: V. Ohnjec (Canadian AWG #014)

ADDRESS: 240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 9xx

PROBLEM:

Proper communication support will be required for Ada tasks when they are distributed over parallel/multi-processor/distributed system environments.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Multi-linguistic, vendor specific run-time system options are available.

POSSIBLE SOLUTIONS:

Modify Ada communication mechanisms to support Distributed/Parallel/Multi-processor environments.

INITIALIZATION OF TASK OBJECTS

DATE: February 2, 1989

NAME: Anders Ardo (Endorsed by Ada-Europe Ada9X working group)

ADDRESS: Dept. of Computer Engineering
University of Lund, P.O. Box 118
S-221 00 Lund, Sweden

TELEPHONE: int+46 46 107522
int+46 46 104714 (fax)
BITNET ddtnet@seldc51
Internet: anders@dit.1th.se
E-mail: anders%dit.1th.se@uunet.uu.net

ANSI/MIL-STD-1815A REFERENCE: 9.1, 9.2

PROBLEM:

It is impossible to provide initialization of a task object when it starts its execution.

Example:

Using a collection of tasks objects of one task type with the intention of having them process different parts of. say, a large matrix. It is impossible to give a task object some initial value or parameter that could indicate which part of the matrix this particular task object should process.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Provide a separate entry that is used only for the purpose of initialization of the task object. Make an entry call to this entry in the newly created task directly after it is created.

POSSIBLE SOLUTIONS:

1. Change the Ada grammar to allow parameters to be specified for tasks (more general).

2. Use discriminants on task types.

PRIVATE TASK ENTRIES

DATE: October 20, 1989

NAME: Thomas J. Quiggle

ADDRESS: Telesoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (619) 457-2700 ex. 158

ANSI/MIL-STD-1815A REFERENCE: 9.1, 9.5

PROBLEM:

Common design paradigms for Ada task interaction require the declaration of entries not intended to be called by tasks external to that owning the entry. The language does not provide a mechanism to guarantee that such entries are not unintentionally called in a manner inconsistent with their intended usage.

One paradigm involves the communication between a task and its lexically enclosed tasks, wherein the enclosing task must be prepared to communicate with more than one of the tasks it encloses (thus preventing the use of an entry call to an enclosed task in lieu of an accept statement). The following example illustrates this form of interaction:

```
task PROCESSOR is
        entry PROCESS_DATA (D : in DATA_TYPE; R : out RESULT_TYPE);
end PROCESSOR;

task CANT_WAIT_FOR_RESULT is
        ...
        entry RESULT_AVAILABLE (RESULT : RESULT_TYPE);
end CANT_WAIT_FOR_RESULT;

task body CANT_WAIT_FOR_RESULT is

        task type AGENT is
                entry SEND(D : DATA_TYPE);
        end AGENT;

        type ACCESS_AGENT is access AGENT;

        TEMPORARY_AGENT : ACCESS_AGENT;
        NEW_DATA        : DATA_TYPE;
        PROCESSED_DATA  : RESULT_TYPE;

        task body AGENT is
                BUFFER : DATA_TYPE;
                RESULT : RESULT_TYPE;
        begin
```

```
                accept SEND(D : DATA_TYPE) do
                        BUFFER := D;
                end SEND;
                PROCESSOR.PROCESS_DATA(BUFFER, RESULT);
                CANT_WAIT_FOR_RESULT.RESULT_AVAILABLE(RESULT);
        end AGENT;

begin -- Cant_Wait_For_Result
        PRODUCE_DATA: loop

                ...-- Produce the data (may involve other rendezvous,
                  -- accept statements for interrupt entries, etc.
                TEMPORARY_AGENT := new AGENT;
                TEMPORARY_AGENT.SEND(NEW_DATA);

        ...-- Code that can't wait for processing of NEW_DATA

        PROCESS_RECEIPTS: loop
                select
                        accept RESULT_AVAILABLE(PROCESSED_DATA) do
                          ...-- Process data returned by earliest returning
                          ...-- agent (which may not be the last agent sent).
                        else
                                        exit PROCESS_RECEIPTS;-- Cannot wait for receipts to
                                                                arrive
                                end select;
                        end select;
                  end PROCESS_RECEIPTS;
                ...
        end loop PRODUCE_DATA;
end CANT_WAIT_FOR_RESULT;
```

In the above example, task CANT_WAIT_FOR_RESULT must continue producing data as its most important activity. Processing of return receipts is of secondary importance, and is performed only when there are receipts waiting. It is possible that several iterations of the PRODUCE_DATA loop will execute prior to receiving any return receipts.

IMPORTANCE: IMPORTANT

Authors of tasks containing entries intended to be called exclusively by lexically enclosed tasks can not guarantee that said entries will not be called externally.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Introduce private task entries that are visible only within the task (including by tasks declared local to the owning task).

```
task_specification :==
        task [type] identifier [is
```

```
        {entry_declaration}
        {representation_clause}
[private
        {entry_declaration}
        {representation_clause}]
end [task_simple_name]];
```

If an address clause is given for a private entry, the entry would be visible to the interrupt source (i.e. the "hardware task" issuing the entry call would be considered local to the task declaring the private interrupt entry). Placing interrupt entry declarations in the private part of a task specification insures that the entry is not called by tasks external to the interrupt task.

PRIVATE ENTRY DECLARATIONS

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 9.1

PROBLEM:

Ada does not provide for private entry points of a task declared in the public part of a package declaration.

IMPORTANCE: IMPORTANT

Often, when several tasks are packaged together, some of their entry points are provided so that users can access features of the tasks and other entry points are strictly for communication between these interrelated tasks.

CURRENT WORKAROUNDS:

Implementors of compilers that are providing specific tasking capabilities with their runtime system simply modify the published (visible) package specification within the implementation library.

Developers must expose all their entries or they must write passthrough procedures or passthrough tasks. The use of passthrough procedures does not allow a user to use task attributes, entry attributes, conditional entry calls or timed entry calls. The use of Passthrough tasks puts an unnecessary extra burden on the runtime system.

POSSIBLE SOLUTIONS:

In a package specification allow task declarations to have a private part. For example:

```
package Task_Master is

    task One is
        entry For_Everyones_Use;
    private
        entry For_Use_By_Task_Two;
    end One;

    task Two is
        entry For_Everyones_Use;
    private
```

```
            entry For_Use_By_Task_One;
        end Two;

end Task_Master;
```

DISTRIBUTED/PARALLEL SYSTEMS

DATE: September 11, 1989

NAME: D. FLemming

ADDRESS: MBB
Dynamics Division
P.O. Box 801149
8000 Munchen 80
West Germany

TELEPHONE: +49/89-6000-5177

ANSI/MIL-STD-1815A REFERENCE: 9.1, 9.2, 9.3, 9.4, 9.5, 9.8, 9.9, 9.10, 9.11, 10.1, 10.2, 10.3, 11.3, 11.4

PROBLEM:

Distributed/Parallel Systems

1. Use of Ada on distributed or parallel loosely coupled processing architectures
   a) Homogeneous processing architectures
   b) Heterogeneous processing architectures
2. Partitioning of Ada programs
3. Allocation of parallel processes (tasks) to processors (construction of static networks)
4. Activation, execution, synchronization, dependence and termination of distributed/parallel Ada units (tasks)
5. Intertask communication (autonomous communication)
   a) Message-driven autonomous communication (autonomous processing of task interactions)
   b) Definition of type-specific messages to control execution of task interactions
   c) Use of modern network standards (e.g., IEEE 802) with powerful individual cast and multicast transfers
   d) Communication model adapted to Ada that harmonize with the proposed autonomous communication concept
6. Memory Management
7. Time in distributed/parallel systems
8. Scope, transfer and identification of exceptions raised

The transfers are controlled autonomously - after initiation of the transfer - by the channels of the communication system.

In the distributed/parallel architecture all messages are generated by sending them (entry call of producer). The contents of messages transmitted are available to the producer even after transmission. Changes done at a later point of time by the producer do not have any effects on messages transmitted.

Receipt of a message (accept statement) means that the message contents is copied by the consumer/s.

Consuming and/or destructive receiving methods destruct the message upon receipt, conserving methods, however, do not. In the latter case messages sent stay in the communication channels of the transport system, are administrated there and only cleared by specific operations. The suggested solution makes use of such a modern method of communication/transport. This method permits asynchronous access of several consumers to messages and thus enables time-efficient execution of multicasts in distributed/parallel architectures.

The channels of the conserving message-transport system autonomously administrate the messages based on message types. Differentiation is made between the type classes "system messages" and "data messages" whose types are basically treated (administrated) differently.

System messages, for instance, are synchronization acceptances or calls, exception handling calls and abort requests. The transport system routes the system messages to the consumer as soon as they are dispatched. Upon acceptance by the consumer the messages are cleared.

Data messages are objects which are evaluated and/or processed by the consumer. The conserving transport system administrates the messages and the access privileges of the consumers according to types.

In general, the suggested solution differentiates between types with:

- multiple access privilege: STATE data
- single access privilege: EVENT data
- lifetime-restricted privilege: EXPIRATION data

Appendix A provides a detailed description of the types and the access mechanisms.

The communication system outlined can be implemented as a pure hardware structure (operating system functions realized in silicon := optimum structure for hard real-time applications) and as a function-identical software structure (distributed operating system functions := structure for complex and large applications).

Distributed/parallel architecture requires that each parallel program unit (task) be assigned a separate processor node with the necessary communication interfaces.

The distribution of several tasks to a processor node (distribution units) is to be supported as well. As far as this kind of distribution is concerned, the communication system differentiates between:

- Node-external message transfers. This means task interaction to tasks located in other processor nodes.

- Node-internal message transfers. This means task interactions between the tasks of a distribution

9. Asynchronous events.

IMPORTANCE: ADMINISTRATIVE

Ada provides no way to design and program distributed/parallel systems. A standardized language support is required in the future. The standard must be based on a correction of ANSI/MIL-STD 1815A and shall specify a way to program distributed/parallel structures of homogeneous or heterogeneous processing architectures in the same manner.

CURRENT WORKAROUNDS:

Data processing under realtime conditions more and more needs computer systems (e.g., embedded systems) realized in static distributed/parallel architecture. Extra-linguistic, project-unique workarounds are currently used to implement Ada applications in this structure of homogeneous or heterogeneous processing architectures.

POSSIBLE SOLUTIONS:

Real-time-critical, security-critical (redundant) and complex applications in avionics, defense systems and automation technology to an increasing extent need static distributed/parallel architectures. For this application domain programming opportunities in Ada are to be created.

An ANSI/MIL-STD 1815A correction and an additional language expansion formed with pragmas may contribute to an acceptable solution.

This solution is to support the compilation of Ada programs for target computers structured as single machines or as static distributed/parallel architectures. The evaluation or ignoration of the pragmas should permit Ada programs to be generated for both architectures.

STATIC DISTRIBUTED/PARALLEL ARCHITECTURE:

Intertask communication is the key to realizing distributed/parallel architectures determining the efficiency of these architectures in computer systems. A suggested solution is outlined in Appendix A. This suggestions are mainly aimed at homogeneous processing architectures and are based on the following structural and runtime properties:

a) Static reconfigurable architecture.

b) Loosely coupled network.

c) Definition of types for task-interaction messages

d) Autonomous transfer of task-interaction messages via the communication channels.

e) Use of modern network standards (e.g., IEEE 802) with powerful individual-cast and multicast transfers.

All intertask communications are executed autonomously by the channels of the communication system in a "producer-consumer" cooperation form. So, message transfers are always unidirectional; they are transmitted asynchronously according to the "no-wait-send" principle.

unit on a processor node.

The node-internal message transfers are to be handled by a local communication manager in the same way with respect to functions as the node-external transfer actions.

Distributed/parallel architectures based on the concept outlined are to be designed by transparent assignment of hardware (processors and communication channels) to software (tasks and task interactions). The basis for this requirement is to be provided in Ada.

ADA CORRECTION AND EXPANSIONS:

The communication model distributed/parallel architectures is to be described in Ada. The hardware structure is to be derived transparently from the communication model. Appendix B, fig.1, shows the problems arising. There must be isomorphism in the functional and real-time behavior of the defaults set by the software and the distributed/parallel architecture realized by the hardware. In this context the allocation of tasks to hardware (sensors, effectors, etc.) within physical network structures is to be taken into consideration.

Without wanting to discuss language constructs in detail it can be said that the following language elements are necessary for the programming and transparent allocation of hardware to software in distributed/parallel architectures:

a) Modularization

Modularization mechanisms must make transparent all import and export functions of tasks (parallel program units) at an interface (task specification). This interface is used to derive application-specific distributed/parallel architectures.

The internal structure, e.g., the implementation of functions related to an interface, is to be protected. This capsulation restricts the transparency areas of data and reduces the dependence of the modules on each other. In this way distribution is guaranteed.

b) Configuration

Configuration is an important aspect in the design of distributed/parallel architectures. Language elements or procedures are required which support the topographic distribution of modularized programs. The location (allocation) of tasks (parallel program units) within the physical network structure of distributed/parallel architectures is to be defined.

c) Concurrency

Language constructs are necessary to mark parallel program units (tasks).

d) Communications (task interactions)

Communications are to be described by means of language elements documenting data-oriented and/or action-oriented transports of messages between parallel program units (tasks). Individual casts and multicasts must be describable.

e) Fault Tolerance

Distributed/parallel architectures must be fault-tolerant so that the processes (systems) they control do not get out of control in case parts of the architectures break down. According to today's state of technology error handling and the necessary redundancy should explicitly be handled by software and not by internal system automatisms. In this way the runtime behavior of architectures becomes clear and predictable and verification gets simpler.

f) Real-time

A program structure is required whose real-time behavior can be verified in the design and development phase of distributed/parallel architectures. The programming language must support the implementation and adaption of the programs and/or parts of the programs for distributed/parallel architectures to the realtime requirements of their applications by transforming (combining/partitioning) parallel program units (tasks).

The real-time clock should be installed centrally in an application of the distributed/parallel architecture. The real-time is to be derived from a system clock interrogated at a processor node of the architecture.

g) Hardware Structure

Restrictions by the hardware structure (processors, memories, signal adaption capacity, etc.) must permit to be planned in the design and development phase. Therefore the programming language must be transparent and flexible.

In particular, parallel program units (tasks) and their interactions must permit to be constructed and documented as transparent allocations of hardware to software. The program structure must permit modularization and documentation according to the requirements of the application; parallelity (concurrency) to its communication embedded in the program must be describable.

The course of design and development of distributed/parallel architectures of the concept outlined is depicted in Appendix B, fig. 2.

In order to realize the concept the following parallel program-units-related Ada corrections and/or supplements may be required:

Problem	ANSI / MIL-STD 1815A	Correction	Expansion
a) Modularization o Import/export functions	available Description not sufficient	-	-
o Task types	Semantics not transferable	Change of semantics	Pragma to define import export functions
b) Configuration	not possible	-	APSE tool for configuration and network construction
c) Concurrency o Activation o Execution o Synchronization o Priorities o Dependence o Termination	available semantics not transferable	- Change or expansion of semantics	
d) Communication o Entries o Entry calls o Accept statements	available Semantics of Ada rendezvous not transferable	- Change or expansion of semantics: o Aysnchronous communication concept o Cooperation form "Producer-Consumer" o Autonomous message-driven communication system	
		Syntax restriction: Communication mode IN is permitted	Pragma to define message types (information to communic. system to administrate messages)
o Abort statem. o Raise statem. o Exception handling	Semantics not sufficient	Correction of semantics	
e) Fault Tolerance	-	less implicit error-trigger mechanisms during runtime	-
f) Realtime	Requirements met due to corrections, expansions listed under a: through e)		APSE tool for runtime analysis
g) Hardware Struct. o Compl. units o Subunits of compil. units o Order of compilation	- unclear	- Tasks are independent compilation units	-

It has to be noted that the suggested solution statically allocates the processors of distributed/parallel architectures to the parallel program units (tasks). This method is in accordance with embedded computer systems in which programs generally are stored in EPROMs. Thus tasks in distributed/parallel architectures cannot be configured and/or allocated dynamically at runtime.

Any incompatibilities of the suggested solution with the ANSI/MIL-STD 1815A are to be investigated and considered when realizing the suggested solution.

Compatibility of this suggestion with nested tasks, access types and the pragma SHARED specified in ANSI MIL-STD 1815A is also still open and requires settlement.

The language corrections and expansions listed are to be adapted to ANSI/MIL-STD 1815A in a way the programs can be compiled both for distributed/parallel architectures and for target computers in single-machine structure. Therefore language expansions are to be realized as pragmas which are effective with distributed/parallel architectures and ineffective with single machines.

PROGRAM DEVELOPMENT SYSTEM:

Appendix B, fig. 3, shows a possible course of development for distributed/parallel architectures based on the suggested solution. The development phase comprises the compilation, runtime analysis, configuration (transparent allocation of hardware to software) and code generating phases and the subphases task separation, task linking and loading.

An APSE is to be used as a development system. The APSE compiler is preceded by a pre-checker which reports Ada programming constructs and/or mechanisms (e.g., pragma SHARED) that cannot be executed on distributed/parallel architectures. The compiler is succeeded by the configurator (allocator) and the runtime analyzer. Based on the lists of network topography descriptions devised by the configurator the distributed/parallel architectures can be designed for programs compiled and the parallel program units (tasks) can be allocated to the processor nodes.

APPENDICES:

1. Appendix A: Paper Reprint of AGARD-Conference Proceedings No.439 "SOFTWARE ENGINEERING AND ITS APPLICATIONS TO AVIONICS". Title of paper: "TASK INTERACTIONS IN DISTRIBUTED MACHINES OF EMBEDDED COMPUTER SYSTEMS" Authors: D. Flemming and H. Grundner

2. Appendix B: Figure 1: Problems of distributed/parallel architectures

   Figure 2: Structures of Real-time Systems

   Figure 3: Programming and Configuration Environment

TASKING ATTRIBUTES APPLIED TO THE MAIN PROGRAM

DATE: June 12, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 9.1 #1

PROBLEM:

Ada tasking model is inappropriate for embedded systems with exact time line boundaries. The model even treats the main program as a task whether there are other tasks or not.

IMPORTANCE: IMPORTANT

The tasking model adds overhead even if it is not used by creating a task activation record for the main program, providing the runtime support, and adding implementation dependencies for abort/terminate/run-forever aspects of a main program.

CURRENT WORKAROUNDS:

Not to use tasking, then customizing the compiler vendors runtime support and delete all undesired overhead functions through special linker controls.

POSSIBLE SOLUTIONS:

Eliminate language in the LRM that would lead a vendor to decide that the tasking model is needed for main programs. It is not enough to say the language is silent on how a main program's exceptional processes are handled or that it is silent on multiple mains. (What does the latter do to shared utilities, multiple interrupt definitions, multiple runtimes, and "the program library"?)

STORAGE_SIZE SPECIFICATION FOR ANONYMOUS TASK TYPES

DATE: October 9, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 9.1(2), 13.2c(10), A(4)

PROBLEM:

The restriction on storage_size specification for task types only causes inappropriate type definitions.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Define a type and one (and only one) object of that type.

POSSIBLE SOLUTIONS:

Allow 'base of a task object so as to permit constructions such as:

```
task X;
for X'base'storage_size use <number>;
```

Alternately, allow storage size specifications for a task object iff that object has an anonymous base type.

ACCESSING A TASK OUTSIDE ITS MASTER

DATE: August 21, 1989

NAME: William L. Wilder

ADDRESS: Naval Sea Systems Command
Department of the Navy
Washington DC 20352-5101

TELEPHONE NUMBER:

ANSI/MIL-STD-1815A REFERENCE: 9.2, 9.4

PROBLEM:

There is a case, specifically permitted by the LRM and by AI-167, in which a task can be accessed outside its master. This one anomaly causes an execution storage problem that precludes the use of dynamically created tasks in long running programs. The anomaly should be eliminated from the language.

IMPORTANCE:

Dynamic task creation cannot be used in long running programs.

POSSIBLE SOLUTIONS:

This is a known problem and is well documented in an ARTEWG ARR submitted by D.Stock (AW-00007). The purpose of this submission is to lend added weight to the requirement as experienced by the ALS/N development team.

In order to provide for the anomaly all ALS/N implementations preserve a residual object for all tasks after termination, forever. This clearly is undesirable for cases of long running programs since storage will eventually become exhausted if dynamic task creation is utilized. There does not seem to be a reasonable solution as all implementations that the ALS/N development team has tested has the same problem: DEC Ada, CYBER Ada, AdaVAX, DDC Ada (VAX target), Janus/Ada, AdaVantage, etc. Different implementations have different sized residuals, but all are finite.

Proposed Solution

The simplest solution is to say that it is erroneous for a task to be accessed outside its master.

INABILITY TO RAISE AN EXCEPTION WHEN A TIME OUT PERIOD EXPIRES

DATE: September 13, 1989

NAME: Jeff Loh

ADDRESS: Intelligent Choice, Inc.
2200 Pacific Coast Highway, Suite 201
Hermosa Beach,California 90254

TELEPHONE: (213) 376-0993

ANSI/MIL-STD-1815A REFERENCE: 9.3.7

PROBLEM:

Inability to raise an exception an exception when a time out period expires.

For example when the time period at an automatic teller machine expires when the customer has left and forgot to complete the transaction. Consider the following code:

```
begin
 TimerManager.SetTimer(DelayInSeconds);     -- (1)
 Process;

exception
 when TimerManager.TIME_HAS_EXPIRED => DoAlternate;

end;
```

The package TimerManager cannot be written using the current Ada facilities. An exception in a task cannot be propagated out. E.g.,

```
task body Timer is
        Seconds : duration;
begin
 loop
  select
   accept Set(DelayInSeconds : duration) do
    Seconds := DelayInSeconds;
   end Set;

  else
          delay DelayInSeconds;
          raise TIME_HAS_EXPIRED;  -- WILL NOT BE PROPAGATED
      end select;
  end loop;
end Timer;
```

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Use polling by declaring an entry to check the status of the time period. Does not work when the subprogram process contains a huge number of subprograms so that the polling check inserted in becomes impractical. This also violates the principle of information hiding. Worse still, Process may be an imported foreign subprogram which the programmer has no control over. Also, if Process contains a input request, the other tasks may or may not be blocked.

POSSIBLE SOLUTIONS:

TASK TERMINATION

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 9.4

PROBLEM:

9.4(13) does not guarantee to uphold the independence of task execution from other tasks as is implied in 9.0. By allowing the ambiguity of the last sentence, there is an allowance for an implementation to artificially force some inter-task dependence for termination.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None, however most vendors do not force task termination is this case.

POSSIBLE SOLUTIONS:

Define in the language, that tasks in this situation are not required to terminate and are expected to continue processing even though the main program has terminated.

TERMINATION OF TASKS WHOSE MASTERS ARE LIBRARY UNITS

DATE: October 24, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail: madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 9.4(13)

PROBLEM:

The termination of tasks whose masters are library units is not defined by the language. This causes portability problems.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Never use terminate alternatives in such tasks.

POSSIBLE SOLUTIONS:

State that the master of such tasks is the anonymous task that calls the main program.

ENTRY AND ACCEPT MATCHING

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 9.5

PROBLEM:

The Reference Manual does not require that an "entry" point declared in a task specification have a corresponding "accept" with the same name.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

No work arounds in the language exist - this matching requires manual or tool-based checking.

POSSIBLE SOLUTIONS:

Include a requirement to make this check in the Reference Manual.

VERIFICATION OF ACCEPT PARAMETERS

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 9.5

PROBLEM:

The Reference Manual does not require that the parameters of an "Accept" be used within the "Accept" statement.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

No work arounds in the language exist - this check is done manually or some software tool.

POSSIBLE SOLUTIONS:

Include a requirement in the Reference Manual to make this check.

ALLOW EXCEPTION HANDLERS IN ACCEPT STATEMENTS

DATE: October 25, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail: madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 9.5

PROBLEM:

Exception handlers are allowed in all kinds of blocks. Why not in accept statements

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Declare an additional block.

POSSIBLE SOLUTIONS:

Change the syntax of accept_statement in LRM 9.5(2) to

```
accept_statement ::=
        accept entry_simple_name [(entry_index)] [formal_part] [do
                sequence_of_statements
        [exception
                exception_handler
                {exception_handler}]
        end [entry_simple_name]];
```

PRIORITY IN ENTRY QUEUES

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 9.5

PROBLEM:

The old priority inversion problem: this is a vote for allowing the runtime system to let a task accept an entry call from a higher priority task, even though the lower priority task tried to call the entry before the higher one. We can't be preemptive, though, so this would probably have to be completed with some kind of priority ceiling algorithm (cf. Ada Europe'89 proceedings)

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: None (except major algorithm redesign).

POSSIBLE SOLUTIONS:

NESTED ACCEPT STATEMENTS

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 9.5(8)

PROBLEM:

Nested accept statements.

One cannot accept entry calls from subprograms directly nested within a task body.

The rules for accept statements exclude the possibility of having a non-empty task stack at the moment of suspension or rendezvous (accepts within procedures that are nested within the task body are not allowed by the rule given above).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use more tasks.

POSSIBLE SOLUTIONS:

This rather technical issue seems to be an insufficient justification for the rule which severely restricts the applicability of tasks. When time-sliced execution is allowed, tasks need to have stacks of their own anyway, the problem is clearly not hard to solve.

The limitation is rather annoying when one needs recursion in a task, For instance to send each value in a binary tree to the caller request, one needs to write something like:

```
type BUFFER_TASK;
type BUFFER_PTR is access BUFFER_TASK;
task type BUFFER_TASK is
        ----Access for the reading side----
        entry END_OF_STREAM (RESULT: out BOOLEAN);
        --      if RESULT is TRUE, BUFFER_TASK will terminate automatically.
        entry GET (V: out T_NODE_VALUE);
        ----Access for the writing side----
        entry PUT_END_OF_STREAM;
```

```
        --this will be the last rendez-vous the writer.
        entry PUT (V: T_NODE_VALUE);
        --E should be returned upon the next GET request.
end BUFFER_TASK;
        task type PRODUCER is
        --Start iteration
        entry START_PRODUCTION (T   :T_BINARY_TREE;
                                BUFFER  :BUFFER_PTR);
        --Send all node values in T to BUFFER, and terminate.
end PRODUCER;

task body BUFFER_TASK is
        CURRENT  : T_NODE_VALUE;
        END_OF_STREAM : BOOLEAN:= FALSE;
begin
MAIN_LOOP:
        loop
                select
                        accept PUT(V:T_NODE_VALUE)
                        do
                                CURRENT :=v;
                        end PUT;
                or
                        accept PUT_END_OF_STREAM do
                                BUFFER_TASK.END_OF_STREAM:=TRUE;
                        end PUT_END_OF_STREAM;
                end select;

                READER COMMUNICATION:
                        loop
                                select
                                        accept END_OF_STREAM (RESULT: out BOOLEAN)
                                        do
                                                RESULT:=BUFFER_TASK.END_OF_STREAM;
                                        end END_OF_STREAM;
                                        exit MAIN_LOOP when BUFFER_TASK.END_OF_STREAM;
                                or
                                        accept GET (E:out P_EXPRESSION.T)
                                        do
                                        if END_OF_STREAM then raise...;
                                        E:= CURRENT;
                                        exit READER_COMMUNICATION;
                                end GET;
                        end select;
                end loop READER_COMMUNICATION;

        end loop;

end BUFFER_TASK;

task body PRODUCER is
```

```
        BUFFER:BUFFER_PTR;
        TREE :T_BINARY_TREE;
        procedure PUT (T:T_BINARYTREE) is
        begin
                if T = null then return;
                BUFFER.PUT(TREE.NODE_VALUE);
                PUT(T.LEFT);
                PUT(T.RIGHT);
        end PUT;
begin
        accept START_PRODUCTION (T  :T_BINARY_TREE;
                                 BUFFER  : BUFFER_PTR) do
        PRODUCER.BUFFER:=BUFFER;
        TREE   :=T;
        end do;
        PUT(TREE);
        BUFFER,PUT_END_OF_FILE;
end PRODUCER;
```

This solution may have its advantages, but if one's primary goal is simplicity, a much better result could be achieved if nested accepts would be allowed:

```
task type PRODUCER is
        ----Access for the reading side----
        entry END_OF_STREAM (RESULT: out BOOLEAN);
        --      if RESULT is TRUE, BUFFER_TASK will terminate automatically.
        entry GET (V: out T_NODE_VALUE);
        ----Access for the writing side----
        entry START_PRODUCTION (T:T_BINARY_TREE);
        --Send all node values in T to caller upon request to entry GET.
end PRODUCER;

task body PRODUCER is
        TREE :T_BINARY_TREE;
        procedure PUT (T:T_BINARYTREE) IS
        begin
                if T = null then return;
                loop
                        select -- Nested accept
                                accept END_OF_STREAM (RESULT: out BOOLEAN)
                                do
                                        RESULT:=FALSE;
                                end END_OF_STREAM;
                        or
                                accept GET (E:out P_EXPRESSION.T)
                                do
                                        E:= TREE.NODE_VALUE;
                                        exit;
                                end GET;
                        end select;
                end loop;
```

```
        PUT(T.LEFT);
        PUT(T.RIGHT);
end PUT;

begin
        accept START_PRODUCTION (T:T_BINARY_TREE) do
                TREE            :=T;
        end do;
        PUT(TREE);
        accept END_OF_STREAM (RESULT :out BOOLEAN)
        do
                RESULT:=TRUE;
                end END_OF_STREAM;
        --Terminate
end PRODUCER;
```

Which is about half as small as the original code measured in lines, and certainly easier to understand.

Another factor is runtime efficiency which is largely determined by the number of rendez-vous that is needed. No need to remark that runtime efficiency is still an Ada design goal (ARM 1.3(05)), and the efficiency that one might obtain when compilers could map dynamically created tasks on distributed processors is probably even worse than on a single processor for the example given; in this example data flow will be the primary factor and computational speed will be negligible, so the number of rendez-vous determines the efficiency of the example, even if compilers that handle distributed processors all by themselves would be available.

Measured in number of rendez-vous, the latter example does 2/3 of the total number of rendez-vous of the former, and if one would minimize the number of rendez-vous needed (by combining GET and END_OF_STREAM in a single entry) this ratio would become 1/2.

ACCEPT STATEMENT WITHIN SUBPROGRAMS AND PACKAGES

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 9.5(8)

PROBLEM:

An accept statement should be allowed within a subprogram or package that is itself within the body of the task (or task type) that declares the entry being accepted.

If an accept statement is executed by a task other than the task that declared the entry (this could happen if the task declares a child task inside the body and the child task call the subprogram), then TASKING_ERROR should be raised. All Accept_Alternatives in a given Selective_Wait statement should be required to refer to entries of the same (statistically enclosing) task.

This change would free programmers from the monolithic-task-body style of programming that is currently required by the language and allow the use of conventional program decomposition techniques. It would also increase the power of Ada tasking by making possible, for example, a task which traverses a recursive data structure and accepts an entry call at each step of the traversal.

One might imagine that this would introduce a number of implementation difficulties, but most of these (e.g. selective waits will terminate alternatives in frames) are already implicit in the current language design because accept statements can occur within block statements.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

In some cases, this problem can be dealt with (at considerable cost in both efficiency and program clarity) by introducing an auxiliary task.

POSSIBLE SOLUTIONS:

REQUIRE TASKS TO HAVE AN ACCEPT STATEMENT FOR EACH ENTRY

DATE: June 6, 1989

NAME: David F. Papay

ADDRESS: GTE Government Systems Corp.
P.O. Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 9.5(8)

PROBLEM:

Currently the standard does not require a task to have an accept statement for an entry declared in its specification. This seems inconsistent, since a function MUST contain at least one return statement, and subprograms with parameters of mode 'out' must assign a value to these parameters.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Some compilers will issue warnings if there is not an accept statement for an entry.

POSSIBLE SOLUTIONS:

Change the last sentence of 9.5(8) to read:

> "... The statements of a task body must include one or more accept statements for each of the entries owned by the task."

then add to 9.5(20-22) Notes:

> "A task body can contain more than one accept statement for the same entry."

CONSISTENT SEMANTICS FOR TASK PRIORITIES

DATE: August 21, 1989

NAME: William L. Wilder

ADDRESS: Naval Sea Systems Command
Department of the Navy
Washington DC 20352-5101

TELEPHONE NUMBER:

ANSI/MIL-STD-1815A REFERENCE: 9.5(15)

PROBLEM:

The current Ada priority system is inadequate for the needs of certain classes of hard real time systems. Ada language semantics should not preclude a consistent use of priorities in determining the allocation of processing resources to tasks.

IMPORTANCE:

Currently users are either bypassing the Ada tasking model completely or are using their own non-Ada executive. These approaches are both non-portable and often inefficient. At best the current situation provides excuses for implementations to obtain waivers to the use of Ada. The ALS/N recommendation is the third (3) alternative enumerated below as that would be the Navy's actual solution should any of the three alternatives be selected.

POSSIBLE SOLUTIONS:

Proposed Solution

Three possible solutions are proposed with increasing levels of completeness and decreasing flexibility:

1. Remove the language semantics that preclude the consistent use of priorities for scheduling decisions. Most notably the last sentence in 9.5(15) requiring FIFO entry queues should be removed. Section 9.8 of the LRM should also be modified to allow the temporary assignment of priorities higher than that specified by the user (to allow for such algorithms as priority ceiling protocol or priority inheritance). Make it clear that implementations may choose their own algorithm but that it must be specified in Appendix F.

2. Augment (1) above with a secondary standard that specifies one or more well specified scheduling disciplines. Modify the wording in 9.8 to provide for the assignment of "execution priorities" that may be different from user defined values. For example a priority inheritance scheme will raise a task priority if it is currently blocking a higher priority task. Such behavior is to be allowed only for authorized schedulers. A mechanism should also be provided for defining new authorized scheduling disciplines. Particular services or institutions could standardize on one or more particular algorithms for specific application types.

3. Modify the language definition to require that implementations use priority consistently when

allocating resources to tasks.

- task entries should be queued in priority order (and FIFO within priorities),
- accept alternatives in select statements should be chosen on a priority basis (and arbitrarily within priority),
- rendezvous should inherit the priority of the highest priority task in the entry queue, and
- tasks should inherit the priority of the highest priority task in any entry queue, even when not in rendezvous.

Inadequacies In Proposed Solution

It is apparent that there is no clear solution to the problem if only from the fact that three alternatives are presented. Alternative (3) provides the most portability but has the highest negative impact on implementations. The first alternative seems to be contrary to the Ada spirit of standardization but provides little or no impact on implementations.

Alternative two (2) seems most viable as it provides for transition, since one approved scheduler could be the current standard, does not require complex algorithms for non hard real time systems, and allows for local standardizations, e.g. service wide, as the proven usefulness of particular scheduling algorithms emerge.

TIMING IN ADA

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 9.6

PROBLEM:

The syntax/semantics for managing "time" in Ada is more for mainframes and Online systems, e.g., passenger airlines reservations, than it is for embedded systems. The model provided is inappropriate for robotics, mission computers, process control. The syntax/semantics is too application and architecture oriented and should not be stated as such for a standard language. Further, realtime, embedded programs with deadlines and needs for determinism can ill-afford loosely controlled time dependent Waits that might come due in such a way as to cause a process to time-out.

The applications have need for two types of delay, very short (microseconds) and somewhat longer (milliseconds) and short typically means less than TBD clock units for an IO instruction. The short delay is not efficient in Ada when it has to be queued and serviced when it gets to it. This causes an obfuscation in the code to avoid Ada managed "delays".

Also, package calendar is not in keeping with our mission requirements of hours not months and days. A space vehicle might be able to use calender, but a missile cannot. There is no loss of generality to give this portion back to the user to define much in the same fashion as for Package SYSTEM.

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Remove the specifications as to how long/how much granularity from the definition of "time", i.e., not every system "knows" or needs to about days/months/years/hours/minutes/seconds. Do not require that the runtime support provide for those features when they are irrelevant. Also, the compiler runtime support should be forbidden to reset the realtime clock. A "known" system utility is called to determine "time" in application dependent units.

Allow for different granularity of time delays. For times that are shorter than a system call, allow the compiler to insert NOPs to account for the delay.

1. Make 9.6 optional.

2. Change the attributes for "time" in subpara. 9.6 #3, 4.

3. Make PACKAGE CALENDAR an example in subpara. #5,6,7. Place the attribute "types" in SYSTEM PACKAGE. Remove wording from LRM like "at least", "at most". Provide for the response time requirement to be application specific and placed in section F. The delay considerations need to include wording for the maximum time within interrupt/exception/control/dispatcher routines. These may be outside the control of the compilation system.

4. "Time" length attribute is not compatible with most realtime clocks. The application is forced to multiple word storage just to support Package CALENDAR, primarily for validation purposes. Package CALENDAR doesn't contain MILLISECONDS or MICROSECONDS which are more suitable than MONTH/DAY/YEAR. Most realtime clocks in embedded systems won't even know or care about "extra large time delays".

5. Selective "waits" are not in keeping with the needs of deadline scheduling and deterministic processing requirements for embedded processors.

CONSISTENT ACCURACY OF CALENDAR.CLOCK WITH RESPECT TO LOCAL SYSTEM TIME

DATE: May 17, 1989

NAME: George A. Buchanan

ADDRESS: IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

TELEPHONE: (301) 731-8894 ext. 2063

ANSI/MIL-STD-1815A REFERENCE: 9.6

PROBLEM:

The value of time returned by CALENDAR.CLOCK may correspond to the true local system time with varying degrees of accuracy. This may cause implementors of real-time systems or even trusted computing base (TCB) systems to use alternate non-Ada means to get consistently accurate values of local system time.

IMPORTANCE: IMPORTANT

If CALENDAR.CLOCK does not return consistently accurate values of local system time its usefulness is unsatisfactorily compromised in time critical situations, especially in real-time systems. Also, time sensitive functions in TCBs, such as time stamping, may be too inaccurate; and the trustworthiness of TCBs may be otherwise compromised.

CURRENT WORKAROUNDS:

Implementors of real-time systems or TCB systems may be compelled to use alternate non-Ada means to get consistently accurate values of local system time.

POSSIBLE SOLUTIONS:

Require CALENDAR.CLOCK to return consistently accurate values of local system time that is adequate within a delta time for real-time and TCB systems. This delta time could be minimized by implementing CALENDAR.CLOCK with a high priority interrupt. The delta time should be specified in Appendix F as an implementation-dependent characteristic.

DELAY STATEMENT IS DESCRIBED POORLY AND NOT USED IN AGREEMENT WITH ITS SEMANTICS

DATE: July 17, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 9.6, 9.7.1, 9.7.3

PROBLEM:

LRM 9.6: "The execution of a delay statement ... suspends further execution of the task that executes the delay statement, for at least the duration specified by the resulting value."

Problem 1. The preceding statement of semantics clearly ties a delay statement to tasking. But is a delay statement really restricted to appearing within a task? Well, no. A delay statement is also a legal constituent of a 'sequence of statements', RM 5, which allows delay statements to appear also in both package bodies and subprograms. Neither (validated) compiler we tried complained about the existence of a delay statement inside a main program that consisted of a single procedure. (Arguing that the task that is delayed in this case is the ephemeral 'some environment task' that calls every main procedure would be the action of a desperate dilettante.) RM 9.6 thus contains a discrepancy between the semantics it specifies for delay statements and section 5.1.

Using delay statements without necessarily employing tasking is useful, for instance, if one wants to display a series of screens of information without requiring user input to cause the display of the next screen.

Problem 2. The descriptions of timed entry calls and selective waits both employ the term 'delay statement' yet neither construct employs the semantics of a delay statement.

LRM 9.7.1 selective_wait_alternative ::= ... | delay_alternative | ...
LRM 9.7.3 timed_entry_call ::= ... or delay_alternative ...

LRM 9.7.1 delay_alternative ::= delay_statement ...

Now, we all know that delay alternatives have none of the semantics of a delay statement -- despite its appearance in the BNF chart; however, figuring that out is part of the lengthy process initiates into the language must complete before they are educated in the language. If the BNF chart shows the existence of a delay_statement, then folks should be able to assume that a delay_statement is what goes there -- including its semantics.

The BNF charts may have been useful for the designers of the language to communicate with one another, but their current purpose is to serve as one more potential pitfall for users of the manual.

IMPORTANCE:

Until the Reference Manual is cleaned up and made a reasonable document, producing new compilers and maintaining existing ones are going to remain much more difficult tasks than they need to be.

CURRENT WORKAROUNDS:

Develop gurus and refer questions to them.

POSSIBLE SOLUTIONS:

One of the unsettling features of this discussion is that the programs noted above could have been very easily avoided. (1) Move the discussion of the delay statement to chapter 5 with the other executable statements and delete use of the term 'task' in its semantics. (2) Change the definition of delay alternative to:

delay_alternative :: = **DELAY** delay_expression;

Those minor changes would have avoided giving people the impression that delay statements are involved in delay alternatives. The result would have been a marginally easier language to implement. What we have now is a sloppily-organized language definition.

USING DELAY FOR PERIODIC TASKS

DATE: September 15, 1989

NAME: Lee W. Lucas

ADDRESS: Naval Weapons Center
Code 31C
China Lake, CA 93555

TELEPHONE: (619) 939-5219

ANSI/MIL-STD-1815A REFERENCE: 9.6

PROBLEM:

Use of the delay statement for implementing periodic tasks (see example below) has several potential problems:

- o Implementations of the delay expression is not guaranteed to be "atomic"; that is, it may be interrupted after Clock is read, but before the delay is activated, resulting in an unknown and perhaps unbounded extra delay time.
- o Reading Clock may take longer than the value of the delay expression, which renders the timing of a hardware clock available that could be used or this purpose).
- o The implementation may be able to handle one or two highest priority tasks in this manner, but cannot handle more, because of excessive calls on Clock or for some other reason.
- o The expression of periodic tasks in this manner is idiomatic (and idiosyncratic?) making it more likely that logic errors will be made when programming periodic tasks.

A proposal for a Delay_Until statement has been advance to deal with the first and second problems above, but it does not solve the third and fourth problems.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

See code below.

POSSIBLE SOLUTIONS:

(1) Add a standard pragma PERIODIC, whose argument specifies the period in seconds, and (2) allow a delay statement with no delay expression, which is interpreted by the runtime system as a suspend until the next period. This solution could be combined with a pragma SPORADIC (or APERIODIC) to allow easy programming of aperiodic tasks.

DETAILED DISCUSSION:

The normal implementation of a periodic task is as follows.

```
--periodic task with period P seconds
with Calendar; use Calendar;
task body Periodic_Task is
        Period          :       Duration:=P;    -- seconds
        Next_Time       :       Time;
begin
        Next_Time := Clock;
        loop
        Next_Time := Next_Time + Period;
        -- work of periodic task goes here
        delay (Next_Time - Clock);
        end loop
end Periodic_Task;
```

The above proposal would allow implementation of a periodic task as follows.

```
--periodic task with period P seconds
task body Periodic _Task is
        pragma PERIODIC (P);
begin
        loop
        -- work of periodic task goes here
        delay;  -- wait for next period
        end loop;
end Periodic_Task;
```

TASKING SCHEDULING IS NOT ABSOLUTE

DATE: September 13, 1989

NAME: Jon N. Elzey

ADDRESS: ITT Defense Communications Division
Dept. 46212
492 River Road
Nutley, NJ 07110

TELEPHONE: (201) 284-4777

ANSI/MIL-STD-1815A REFERENCE: 9.6 (3), 9.7.1 (5,8), 9.7.3 (3,4)

PROBLEM:

Task scheduling provided by the delay_statement is relative, not absolute.

Some real-time applications require task scheduling at specific absolute times, so that tasks can synchronize with deterministically timed events. Such scheduling is specified by a delay_statement or a delay_alternative.

A computed delay using CALENDAR.CLOCK, as in the LRM example (9.6), yields an unpredictable absolute time for delay expiration because the execution times for CALENDAR.CLOCK, delay simple_expression computation, and run-time system overhead cause slippage beyond the absolute time intended by the delay_statement simple_expression. Contention from higher priority tasks and interrupts contributes to this slippage. In the worst case, an execution deadline may be missed.

IMPORTANCE: IMPORTANT

This problem is of importance only to real-time applications, but these are a major application area for Ada, and one that continues to generate resistance. Solving this problem may help to prevent proliferation of alternative incompatible and non-portable tasking implementations.

CURRENT WORKAROUNDS:

A minimum observed slippage can be included in the delay_statement simple_expression.

If the compiler vendor's run-time system and CALENDAR package can be tailored to service the delay only at some deterministic interval that will support the application, then this interval may be long enough to absorb slippage in the delay computation while still providing predictable absolute times. Given predictable delay expiration times, it becomes easier in the application design to allow for the remaining slippage from delay expiration to the time the task begins to run.

POSSIBLE SOLUTIONS:

Overload the delay_statement so that it also takes a simple_expression of type CALENDAR.CLOCK. This would be a compatible extension of the current standard.

Alternatively, add to the language an until_statement, but specifying an absolute delay expiration time.

CONDITIONAL ENTRY CALL SYMMETRY WITH ACCEPT STATEMENTS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 9.7

PROBLEM:

It should be possible to select any of the following options:

* Accept, if something in the queue
* Entry call, if callee is callable
* Delay
* Null

That way, it would be possible to send messages to a task that is waiting to call other tasks.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

CONDITIONAL ENTRY CALL ON GENERIC FORMALS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 9.7

PROBLEM:

Currently, it is not possible to make a conditional or timed entry call to an entry that has been supplied to a generic through with procedure. This is irritating, since the usage pattern is usually known.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Raise a predefined exception if a conditional entry call is attempted on something that was instantiated with a procedure (or ignore the else branch altogether, just giving a compile time warning). An alternate approach would be to allow with entry generic formals.

ORTHOGONALITY OF SELECT STATEMENTS

DATE: October 9, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 9.7

PROBLEM:

The conditional and timed entry calls are nearly identical to a selective wait with only one select alternative (and no terminate alternative). Since the only difference between an accept statement and an entry call is the direction of the call and the scope of visibility of the critical region, this represents an inconsistency of the language.

IMPORTANCE: IMPORTANT

The lack of ability to call the first available entry causes usage of odd structures and polling.

CURRENT WORKAROUNDS:

A coordinator task or a call-reversing task needs to be used to implement a selective wait on entry calls. An example would be a set of server tasks that report their availability to a dispatcher rather than having the user call the first available one.

POSSIBLE SOLUTIONS:

Extend the definition of the selective wait alternative [9.7.1(2)] to include an entry alternative consisting of an entry call statement and a sequence of statements. The changes would read:

```
selective_wait_alternative ::=
        accept_alternative  |  entry_alternative |
        delay_alternative  |  terminate_alternative

entry_alternative ::=
        entry_call_statement [sequence_of_statements]
```

ASYNCHRONOUS TRANSFER OF CONTROL

DATE: August 29, 1989

NAME: William L. Wilder

ADDRESS: Naval Sea Systems Command
Department of the Navy
Washington DC 20352-5101

TELEPHONE NUMBER:

ANSI/MIL-STD-1815A REFERENCE: 9.7.1

PROBLEM:

Asynchronous transfer of control is needed for real time systems to support fault recovery and mode changes. [See Ada letters special edition Vol VII Fall 88].

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

The Navy has a requirement for asynchronous transfer of control within the ALS/N in order to provide for mode changes and fault recovery mechanisms in Mission Critical Computer Resource (MCCR) systems. Until recently the approach considered was asynchronous exceptions. However, as a result of Navy participation in the 3rd International Workshop on Real Time Ada Issues (IWRT3) the ALS/N community endorses the approach proposed in the Ada 9X revision request 9X-00083/00 by S. Tucker Taft which is based on an extension to the selective wait construct. The remainder of this ARR repeats that ARR or copies materials generated by discussions at the IWRT3.

This approach solves both the time-out and asynchronous control problems within a single construct. It separates the concept of asynchronous transfer of control from exception handling, therefore avoiding the problems introduced by asynchronous exceptions.

Proposed Solution

The proposed approach is to provide an additional selective wait construct

```
select
        select_alternative
{ or
        select_alternative}
and
        sequence_of_statements
end select;
```

The semantics of this construct are that normal processing is performed on the select alternative guards to determine which should be "open." Selection of one such open alternative takes place immediately if a rendezvous is possible, or if a delay alternative of less than or equal to zero seconds is open. Otherwise, the sequence of statements begin execution. If the sequence_of_statements completes execution, then the select alternatives are closed. If prior to completion of the sequence_of_statements and outside of any nested rendezvous (either accept or entry call), a delay alternative has expired, or an open entry alternative has a caller, then the sequence_of_statements is abandoned, and an asynchronous transfer of control takes place to the appropriate select alternative. This abandonment takes place no later than the next synchronization point, but it is the intent that any ongoing computation (outside of a rendezvous) be preempted.

If the same entry is made open via a nested selective wait or accept statement, then the inner construct takes precedence.

Examples Of Intended Use

```
-- command interpreter (from 9X-00083)
loop
  select
        accept Terminal_Interrupt
        Put_Line("Interrupted");
  and
        -- abandon command execution on terminal interrupt
    Put_Line( "->" );
    Get_Line(Command, Last);
    Process_Command(Command(1..Last));
  end select;
end loop;

-----------------------------------------

-- time-limited calculation (from 9X-00083)
select
  delay 5.0
  Put_Line("Calculation does not converge");
and
  -- This calculation should finish in 5.0 seconds
  -- if not, it is assumed to diverge
  Horribly_Complicated_Recursive_Function(X,Y);
end select
```

Alternatives Considered

The primary alternative considered was asynchronous exceptions. The following summarizes some of the problems with asynchronous exceptions

- The asynchronously raised exception can be accidentally handled or converted to other exceptions via handlers for "others" within called subprograms, so that the signal is not recognized;
- the task in which the exception is raised may be blocked at an entry call, select, or accept statement;

- the asynchronously raised exception may be accidentally propagated to other tasks via rendezvous;
- multiple asynchronous raise operations can cause confusing nondeterministic effects;
- eliminates optimizations performed in limited scopes which compiler can determine that no exceptions are raised;
- provision in Ada which ensures that exceptions raised will be handled may be circumvented.

SYMMETRICAL SELECT

DATE: October 23, 1989

NAME: Marc Riese
Giovanni Conti
Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail: riese@litsun.epfl.ch
E-mail: conti@ltisun.epfl.ch
E-mail: madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 9.7.1

PROBLEM:

Currently, selective waits are possible only for accept statements, not for entry calls. This causes problems for communications between servers, in which case one is often obliged to add messenger tasks that would be unnecessary if the selective wait statement were more flexible. See references, especially the paper by Riese and Conti, for a complete study of the problem.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use messenger tasks.

POSSIBLE SOLUTIONS:

Replace LRM 9.7(2), 9.7.1(2), 9.7.2(2) and 9.7.3(2) with the following:

```
select_statement ::=
        select_alternative
{or
        select_alternative}
[else
        sequence_of_statements]
end select;

select_alternative ::=
        [when condition =>]
                selective_wait_alternative
```

selective_wait_alternative ::= accept_alternative |
entry_call_alternative | delay_alternative | terminate_alternative

accept_alternative ::= accept_statement [sequence_of_statements]

entry_call_alternative ::= entry_call_statement [sequence_of_statements]

delay_alternative ::= delay_statement [sequence_of_statements]

terminate_alternative ::= **terminate**;

Thus merging selective_wait, conditional_entry_call and timed_entry_call into a single construct.

One problem with this change is that the note in LRM 9.5(21) saying that "The language rules ensure that a task can only be in one entry queue at a given time" would not be true anymore, but this is only an implementation problem.

REFERENCES:

R. Bagrodia, "Process Synchronization: Design and Performance Evaluation of Distributed Algorithms". IEEE Transactions on Software Engineering, Vol. 15(9), 1053-1065, Sept.89.

N. H. Gehani and T. A. Cargill, "Concurrent Programming in the Ada Language: The Polling Bias". Software Practice and Experience, Vol 14(5), 413-427. May, 1984.

Rob Pike, "Newsqueak: A Language for Communicating with Mice". AT&T Bell Labs, New Jersey. Computing Science Rechnical Report 143. 1989.

M. Riese and G. Conti, "Drawbacks of Ada's Synchronisation Mechanism and a Simple Solution". Position paper for the 3rd Intn'l Ada Real-time Issues Workshop. Pennsylvania, June 1989.

TASKS WITH DELAY AND TERMINATE ALTERNATIVES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 9.7.1(3)

PROBLEM:

Due to a restriction in the standard, this tasking construct is not permitted:

```
select
        when Some_Guard =>
                delay Some_Time;
        or
                ...
        or
                terminate;
end select;
```

The need for such a construct arises fairly often.

IMPORTANCE: IMPORTANT

The workaround for this introduces a maintenance problem, and there does appear to be any valid reason for the restriction to exist in the first place.

CURRENT WORKAROUNDS:

The programmer can write the guard in the form of an IF statement:

```
if Some_Guard then
        select
                delay Some_Time;
        or
                ...
        end select;
else
        select
                ...
        or
                terminate;
```

```
        end select;
end if;
```

Unfortunately, this is error-prone, and introduces a maintenance problem because the two arms of the IF must be maintained in sync with one another.

POSSIBLE SOLUTIONS:

Relax 9.7.1 (3) to allow the currently prohibited construct.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

ENTRY CALLS WITH TERMINATE ALTERNATIVES

DATE: July 25, 1989

NAME: R. David Pogge

ADDRESS: Naval Weapons Center
EWTES - Code 6441
China Lake, CA 93555

TELEPHONE: (619) 939-3571
AUTOVON: 437-3571
E-mail: POGGE@NWC.NAVY.MIL

ANSI/MIL-STD-1815A REFERENCE: 9.7.2

PROBLEM:

There are two ways a program can end in a task deadlock, but only one is handled by Ada. (1) A task can be waiting to accept an entry call, but the calling task has terminated normally. Ada's selective_wait statement includes a terminate alternative that handles this situation. (2) A task can be waiting to call another task, but the called task has been terminated normally. Ada does not have any clean way to deal with this situation. (At least one implementation senses the global deadlock and terminates with an error message.)

IMPORTANCE: ADMINISTRATIVE

This is ADMINISTRATIVE because it makes the standard more consistent and easier to understand, and also IMPORTANT because it gives needed capability. It is not essential, because one can always end the program by crashing, or some other workaround.

CURRENT WORKAROUNDS:

There are workarounds, but they are too long and involved to discuss here. The best one is Solution 4 of the Print Spooler problem.

POSSIBLE SOLUTIONS:

Change the definitions of conditional_entry_call and entry_call_alternative as shown below:

```
conditional_entry_call ::=
  select
    entry_call_statement
   [sequence_of_statements]
  {or
    entry_call_alternative}
  [else
      sequence_of_statements]
  end select;
```

```
entry_call_alternative ::=
  entry_call_statement [sequence_of_statement]
| delay_alternative
| terminate_alternative
```

Note: This change makes timed entry calls a special case of the conditional entry call. The language is more consistent and symmetrical.

TERMINATE NOT USABLE

DATE: October 17, 1989

NAME: Bjorn Kallberg

ADDRESS: Ericsson Radar Electronics
S-164 84 Stockholm
Sweden

TELEPHONE: 46 8 757 35 08
46 8 752 81 72 (Fax)
E-mail kallberg@kiere.ericsson.se

ANSI/MIL-STD-1815A REFERENCE: 9.7.1 (3)

PROBLEM:

The terminate alternative was intended to be a means of getting server task to stop execution, without resorting to special stop entries. (Rationale for the design , chap 13.2.10). One example where this is important is information hiding, to be able to hide the active part of an object from the user.(A library package or a private data type, where the user may not know that there is a task).

Unfortunately, this objective is not fulfilled. The reason is: In practice, many of these servers often a need some cyclic work, even when no calls from clients arrive. This can not be coded today. The naive way of stating the problem is "A selective wait can not contain both a delay and a terminate alternative"

It is also a consequence of the visibility rules, that this problem can not be circumvented by having a cyclic task call an entry in these server task. In such a case, the terminate alternative would never be chosen, as the cyclic task is not completed.

Thus, the intended elegance of the terminate alternative can only be utilized very seldom.

Another part of the problem is that implementation of the terminate alternative is difficult, and have a negative impact on tasking performance

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Do not use the terminate alternative. Have a special stop alternative in each task, have the main program to know the names of each stop entries to call, and call them. However, this is not so simple as it may seem, as the tasks must be stopped in proper order, tc avoid the "tasking error" exception. The main program must thus also know the dependence between these tasks.

POSSIBLE SOLUTIONS:

Make the termination user controlled, but without need for the user to name and know each individual task.

Below is a possible solution along those lines: Define a general, anonymous accept alternative (say "newterminate") which can be used in every task. This alternative can only be accepted after an explicit call ("startterminationn") made by some task or the main program. One single such call applies for all newterminate alternatives.

After this call has been given, a task terminates if it has no dependent tasks and is waiting at a selective alternative with a newterminate alternative. After this call has been given, no delay alternatives are taken in any selective wait which also contain a newterminate alternative.

This "startterminationn" call is given when the intent is to stop the whole system. Before the call, the newterminate alternatives are effectively non existent. After the call, the system is not in any time-critical phase, and the implementation may take its time.

SCHEDULING ALGORITHMS

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
e-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 9.8

PROBLEM:

Need to be able to specify the use of scheduling algorithms more sophisticated than fixed priority, pre-emptive scheduling, e.g., priority inheritance, ceiling protocol.

IMPORTANCE: ESSENTIAL

Ada will otherwise not be used for systems with hard real time constraints.

CURRENT WORKAROUNDS:

Non-standard, unvalidated Ada run time systems.

POSSIBLE SOLUTIONS:

One would expect the use of priorities by the programmer to be replaced by some way of specifying scheduling requirements in terms of deadlines and relative importance, with some ability to vary relative importance at run-time to cope with changed modes of operation. Ideally there would be some pre-defined type named scheduling_requirements, but I doubt whether there is yet sufficient consensus for this, so it would have to be implementation defined.

ALLOW DYNAMIC PRIORITIES FOR TASKS

DATE: August 21, 1989

NAME: William L. Wilder

ADDRESS: Naval Sea Systems Command
Department of the Navy
Washington DC 20352-5101

TELEPHONE NUMBER:

ANSI/MIL-STD-1815A REFERENCE: 9.08

PROBLEM:

The user must be provided with the capability of modifying task priorities dynamically. The language should not preclude the explicit modification of priorities by the application.

IMPORTANCE:

Applications and implementations will go outside of the language to resolve this issue. The ALS/N position is to provide a procedure for a task, not assigned a priority, to change the priority of itself. Ada semantics will be followed in that the LRM allows the implementation to define scheduling order where priorities have not been defined.

POSSIBLE SOLUTIONS:

Dynamic priorities are necessary for systems which undergo mode changes or need to provide for degradation in computing resources. Dynamic priorities are useful in other situations where workarounds are possible but awkward.

It is proposed that a new attribute ('PRIORITY) be introduced and a new procedure (CHANGE_PRIORITY) provided in package SYSTEM. These new facilities will allow a task to interrogate and change its own priority value. If a facility is provided in 9X for the identification of tasks (not proposed or recommended here), then the ability of one task to change the priority of another task should be considered.

Proposed Solution

The new attribute T'PRIORITY is provided which returns the current priority of the designated task object T. The value returned is of type SYSTEM.PRIORITY. For tasks that have not been explicitly assigned a priority via pragma or otherwise, the implementation will return whatever priority is being used by the scheduler.

It is considered vital that a task be allowed to change its priority value in addition to setting an initial user defined priority. It is therefore proposed that a procedure be provided in SYSTEM:

```
procedure CHANGE_PRIORITY( P : SYSTEM.PRIORITY );
```

There have been proposals to provide for task identifications (see e.g. ARTEWG), or to in some other way raise the status of a task object to a full fledged Ada object. If such a mechanism is provided in 9X, and such a mechanism is NOT being recommended here, then the ability of one task to change the priority of another task should be considered. The semantics of such a facility needs to be clearly defined since scheduler queues will likely be affected.

Examples of Intended Use

It is typical for defense systems to have several modes of operation. For example, BMEWS has a general surveillance mode which includes ground or sea launched ballistic missile and satellite tracking. On launch, a determination may be made as to where the projected impact point is, indicating a test launch or a potential attack. Completely different priorities of tasks are warranted for a potential attack versus continued surveillance.

Other examples can be found in ARTEWG ARR AW-00009/00 expected to be submitted.

Inadequacies In Proposed Solution

The change priority of self proposal is particularly simple to implement and has little impact on existing tasking semantics. This is derived from the fact that a currently executing task is not in any queue. The change priority of another task has ramifications since it will probably require a reordering of the current scheduler run queue, but perhaps not if the target task is in rendezvous with another and has inherited that task's priority. Also, if the target task is the client task in rendezvous, the server task may need to have its priority changed also, which could be the client of a third task, etc. Another potential complication could arise if 9X adopts or allows priority ordered entry queues. In this case a change in priority could cause a reordering of an entry queue as well.

Alternatives Considered

Similar results could be achieved by altering the wording in the 9X LRM to clearly allow for an implementation providing a mechanism for the dynamic modification of priorities. (Some currently interpret the language definition to allow this currently.) In conjunction with this relaxation a distinct package containing a subtype PRIORITY, CHANGE_PRIORITY and perhaps GET_PRIORITY could be introduced as a secondary standard, rather than have them inside SYSTEM.

IMPROVED SUPPORT OF PRIORITY LEVELS

DATE: September 26, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@TREES.DNET.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 9.8

PROBLEM:

The current Ada language specification provides very limited support for priority levels. For example:

1. There is currently no way to dynamically change priorities. For example, the priority of a task measuring a variable should have increased priority of the value approaches its legal limits. (For example, measuring the altitude of an aircraft becomes much more important as the plane descends from twenty thousand feet to twenty feet.)

2. In real situations, the priority of a task is often dependent on how long it has been since the task was previously executed. There should be a means of arranging for the priority of a task to be automatically raised as time goes by.

3. It may be desirable to have a task run with different priorities depending on the entry used to rendezvous with the task.

IMPORTANCE: IMPORTANT

Although it might impose some difficulty in implementation, the improved support of priority control should aid in the construction of robust systems.

CURRENT WORKAROUNDS:

It becomes very difficult to arrange workarounds for this problem. One possibility is to assign very high priorities to tasks and then effectively force the priorities down by the use of delay statements inserted in the code.

POSSIBLE SOLUTIONS:

It might be possible to have a procedure with the calling sequence

```
SET_PRIORITY (CURRENT_PRIORITY: in PRIORITY)
SET_PRIORITY (CURRENT_PRIORITY: in PRIORITY;
              NEXT_PRIORITY: in PRIORITY;
              ACTIVATION: in DURATION)
```

In the first case, CURRENT_PRIORITY is the value to be used for the priority, with this level of priority to be used until explicitly changed. In the second case, the priority is set to CURRENT_PRIORITY, but it will then be reset to NEXT_PRIORITY after a period of DURATION. Each instance of SET_PRIORITY will override the values set by the previous call of the subroutine.

In addition, there should also be a set of subroutines to read the values set by the SET_PRIORITY procedures.

```
function PRIORITY return PRIORITY
```

would return the current priority level while

```
GET_NEXT_PRIORITY (NEXT_PRIORITY: out PRIORITY;
                   ACTIVATION: out DURATION)
```

would return information on the next priority to be used.

TASK PRIORITIES, PROCESSING OF ENTRY CALLS

DATE: September 27, 1989

NAME: K. Buehrere

ADDRESS: ESI
Contraves AG
8052 Zuerich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 9.8, 9.7.1, 9.5

PROBLEM:

A considerable and increasing number of problems with Ada tasking have been identified since Ada is used in real-time applications. Most of them arise because of the priority rules, the scheduling rules or the way entry calls are processed. Please consider the following examples:

- The priority of a server task (executing and accept statement) is raised to the priority of its client (executing an entry call) during the rendezvous only. Outside the rendezvous, the server does not "feel" the priority of any waiting client. This may lead to the well-known phenomena of "priority inversion".

- Incoming entry calls are processed in their order of arrival, which is not always convenient. In some cases, entry calls of high priority tasks should be handled before entry calls of low priority tasks, no matter the order of arrival.

- When a task executes a selective wait statement with several open accept alternatives, it is free to accept any of the waiting callers. In some applications, however, a waiting caller should be selected in a defined way, and not just "arbitrarily". It would be reasonable to select e.g. the caller with the highest priority or the caller already waiting for the longest time.

IMPORTANCE: ESSENTIAL

These and other problems with tasks and priorities have already been described in the literature. It seems to be rather difficult to use Ada tasking in real-time applications - the field Ada was originally designed for! This handicap may well delay or reduce the acceptance of Ada and thus threaten the overall goals of introducing the language.

CURRENT WORKAROUNDS:

No general workaround is available.

Specific problems may be solved through a careful layout of task interaction. Entry families may be used to simulate priority sensitive processing of entry calls. However, any workaround is in general clumsy and does greatly impair run time efficiency.

POSSIBLE SOLUTIONS:

. Priority-Inheritance
If the pragma

PRAGMA inherit_priority(base_priority => some_value);

appears in a task specification, this task will inherit the priority from its waiting callers. I.e. the priority of the task will never be lower than the priority of any of its callers, no matter whether currently engaged in a rendezvous or not. The pragmas priority and inherit_priority are mutually exclusive, the parameter of inherit_priority is optional. Tasks with a fixed priority (pragma priority) and task inheriting priority may be mixed in one program.

. Priority_Sensitive Entry Queues
A pragma, say

PRAGMA priority_queue(entry_simple_name);

may be applied to an entry within a task specification this would require, that incoming calls had to queued and processed in the order of caller-priorities and not in the order of arrival. Other entries of the same task may still queue their calls in the order of arrival.

. Priority-Sensitive Selective Wait
If a pragma, say

PRAGMA priority_select;

appears a (first?) select alternative in a selective wait statement, waiting callers are selected according to their priority, and not arbitrarily. Of course, the current definition of the language already allows this behavior, but does not require it.

The suggested extensions are at least upward compatible. No negative impact on existing implementation is to be expected, even if implementation of priority-inheritance may be complicated.

DYNAMIC PRIORITIES

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 9.8

PROBLEM:

Task priorities must be assigned through explicit constants. This means that it is not a simple integration-time exercise to balance the relative priorities of tasks.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Using direct OS calls (which usually makes life tough for the runtime system)

POSSIBLE SOLUTIONS:

Provide a priority setting primitive.

ABORT IS USELESS

DATE: March 3, 1989

NAME: Anders Ardo (Endorsed by Ada-Europe Ada9X working group)

ADDRESS: Dept. of Computer Engineering
University of Lund, P.O. Box 118
S-221 00 Lund, Sweden

TELEPHONE: int+46 46 107522
int+46 46 104714
BITNET: ddtent@seldc51
Internet: anders@dit.1th.se
E-mail: anders%dit1th.se@uunet.uu.net

ANSI\MIL-STD-1815A REFERENCE: 9.10

PROBLEM:

The semantics of abort is too loosely defined on the LRM. This can give rise to large discrepancies in different implementations and thus cause portability problems.

Quote from LRM (9.10 par 6):

> "The completion of any other abnormal task NEED NOT happen before completion of the abort statement. It MUST happen no later than when the abnormal task reaches a synchronization point..."

This means that an aborted task can happily go on and execute until it reaches a synchronization point, which maybe never happens OR die instantly depending on a particular implementation. This is not satisfactory.

The restriction (in 9.10 par 6) for abort not to interfere with a rendezvous in action is unfortunate (however understandable) since it excludes the possibility of unconditional termination in certain situations.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Do not use abort

POSSIBLE SOLUTIONS:

As abort is defined today, it is virtually useless. Either the semantics has to be made more binding without special cases and possible indeterminate delays, OR the abort statement should be removed from the language.

COMPLETION OF TASKS THAT ABORT THEMSELVES

DATE: June 1, 1989

NAME: David F. Papay

ADDRESS: GTE Government Systems Corp.
P.O. Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 9.10(5-6)

PROBLEM:

The current language standard allows a task that names itself in an abort statement to continue executing until the next synchronization point is reached. I feel that the intent of the language was for such tasks to become complete immediately after executing the abort statements, and this should be explicitly stated in the standard.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

If a task names itself in an abort statement, and this name is static, then a delay statement can be coded immediately after the abort, forcing a synchronization point. However, tasks named in an abort are not always static (i.e., aborting a task in an array of tasks where the array index is a non-static), and this workaround would force the delay to be executed whether or not it was actually needed.

POSSIBLE SOLUTIONS:

Change the wording of 9.10(5):

> "...This completes the execution of the abort statement. If the task that executed the abort statement was itself made abnormal by that statement, it becomes complete immediately after execution of the abort statement has been completed."

ATOMIC READ/WRITE OPERATIONS FOR SHARING VOLATILE MEMORY

DATE: October 1, 1989

NAME: Ivan B. Cvar, Canadian Ada Working Group

ADDRESS: PRIOR Data Sciences Ltd.,
240 Michael Cowpland Drive,
Kanata, Ontario, Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 9.11

PROBLEM:

There is a need for a mechanism that allows an Ada program to atomically read and write to volatile memory, particularly when memory must be shared between hardware devices or between multiple processors that communicate via a single memory block.

This is a common requirement of embedded systems, which interface more directly to the hardware than do other types of applications.

IMPORTANCE: *ESSENTIAL*

CURRENT WORKAROUNDS: NONE

Pragma SHARED is insufficient for this purpose since it applies to different tasks in the same program, and also because it is not supported by some compilers.

POSSIBLE SOLUTIONS:

Perhaps a new predefined pragma VOLATILE could be introduced to allow read and write "locks" to be inserted in locations known to all sharing the memory.

It was recognized that the resulting program would be target dependent and less portable, but it was felt that the benefits for an embedded system would far outweigh the costs. It was further noted that some current hardware architectures don't yet support guaranteed atomicity, but as multiprocessing becomes more available, the need for atomicity becomes essential, and its hardware support will follow.

ASSIGNMENT TO BE AN INDIVISIBLE OPERATION

DATE: October 27, 1989

NAME: Jan Kok (on behalf of the Ada-Europe Numerics Working Group)

ADDRESS: Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam-NL

TELEPHONE: +31 20 5924107
+31 20 5924199 (fax)
E-mail: UUCP: jankok@cwi.nl

ANSI/MIL-STD-1815A REFERENCE: 9.11

PROBLEM:

There is no guarantee that assignments onto shared objects of user-defined types like a type COMPLEX or composite and private types in general, or, even more seriously, onto shared objects of scalar types, are treated as indivisible operations, when 'whole object' updating is performed in a distributed architecture. The workaround of programming a synchronization point before and after complete updating is complicated and unnecessarily verbose. The user should have a tool to request this indivisibility for all 'whole object' operations on objects of a given type.

Considering that for objects of an arbitrarily complex structure it may be impossible to obtain such an indivisibility of the assignment, we require the indivisible assignment for scalar and access types only.

IMPORTANCE: ESSENTIAL

This requirement is actually ESSENTIAL for everyone as it involves the integrity of data written and read.

CURRENT WORKAROUNDS:

Through explicitly programmed synchronization points.

POSSIBLE SOLUTIONS:

The requirement can simply be added to the semantics.

PRAGMA VOLATILE

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 9.11

PROBLEM:

Pragma Shared is not sufficient for the purposes to which C's volatile can be used. It is only intended for the sharing of variables between tasks of the same program. Several additional semantics are needed:

- Every read or write of this variable or this component of a record must reference memory immediately.
- This variable must be correctly represented in memory at this point in execution.

Certain functions cannot be distributed without these semantics.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Write something in assembly language, either through Pragma Interface or a machine code insertion.

POSSIBLE SOLUTIONS:

Permit a Pragma Volatile, which can be applied to a type, an object, or component of a record object, indicating that all reads and writes of the object or component must reference memory. Permit a Pragma Update which can occur where a statement can occur, and indicates that the value of its argument object must be in memory before the next statement is executed.

The atomicity of the access is probably not critical, but the thing which is actually volatile could probably be restricted to a scalar, although it must be possible to have a record with a volatile scalar component.

A representation clause might be a better idea for Pragma Volatile, since a pragma can be ignored, and ignoring Pragma Volatile (or Pragma Shared) changes the semantics. Additionally, the presence of this requirement may restrict the representations available, due to interactions between atomicity and alignment. To use something other than a pragma for Update, we would need something like a procedure call with a reserved name, for example "access<object_name>;".

PRAGMA SHARED

DATE: June 27, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 9.11(9)

PROBLEM:

Pragma SHARED is too restrictive.

When implementing reference counting memory management (e.g. for symbol tables or LISP expressions), one typically implements objects as pointers to records that have a subcomponent of an integer type.

This approach works just fine until one starts passing these values among tasks. At that moment the approach seems to be a great solution because of the functional programming style that avoids value updates and hence seems to avoid all problems related to mutual exclusion between synchronization points as explained by 9.11.

Reality is different though, since the reference counts get updated each time the object gets shared by a reference more or a reference less. This situation poses the programmer for a very difficult situation : either reference counts \need to be implemented by dynamically created tasks that protect a reference count or one needs to copy instead of share. Protecting implies very severe limitations on almost any compiler. For example:

* a minimum storage requirement of 1Kbyte per object
* about 60 microseconds for each increment/decrement operation
* a global limitation of 500 objects

and this is supposed to be the high quality compilers, so these figures are far away from the worst case.

The other alternative is copying instead of sharing. In case of a symbol table this defeats the whole purpose of the package, in case of LISP expressions this will slow down the execution of non-trivial LISP interpretation to unacceptable levels.

When probing Ada experts about this topic one is likely to get pragma SHARED for an answer.

The problem that remains is twofold: pragma SHARED is only allowed for variables and not for record components, and increment and decrement are not considered indivisible operations.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Make copies only when passing such objects from one task to another (slow and error-prone), use only tasks that will not interfere with the reference counts (less parallelism since a master needs always to wait until the slave did his work in order to clean-up the object if needed, also error-prone), or don't use such types in entry calls at all (destroys the tasking benefit).

POSSIBLE SOLUTIONS:

Allow pragma SHARED to apply to record components, and support procedures INCREMENT and DECREMENT in package STANDARD for all scalar types.

The only real problem with pragma SHARED is referencing the record component, implementing the feature can't possibly be a problem since one could always introduce a semaphore as a hidden record component.

The reference to the component might be a long name starting with the type name and using selection to indicate subcomponents, or one could allow pragma SHARED nested in record type declarations (immediately following a component declaration), or one might allow pragma SHARED in record representation clauses or add an optional SHARED keyword in the component clause.

Supporting INCREMENT and DECREMENT as basic operations shouldn't be any problem at all, and at the same time this would also prevent ex-C-addicts to get delirious about the lack of the ++ and -- operations, which, let's admit, are not so dirty at all if they wouldn't have been defined as side-effect functions. Surely one can write these procedures oneself, but since the overhead of writing them, importing them or instantiating them is typically larger than the benefit, I guess that at least 99% of the INCREMENT (expression); operations are written as expression := expression + 1;.

For additional references to Section 9. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 10. PROGRAM STRUCTURE
AND
COMPILATION ISSUES

USE OF LARGE NUMBERS OF LIBRARY UNITS CAN GIVE RISE TO NAME CLASHES

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 10

PROBLEM:

When writing large systems in a modular fashion, with existing compilers, each module has to be a library unit, for two reasons. Firstly, neither subunits nor parts of compilation units can be reused without source code changes. Secondly, extensive use of library units, each with its own specification, reduces the amount of recompilation needed when changes are made (note that with current compilers, just adding a procedure to a package causes recompilation of all units that have a "with" statement for the package).

However, use of large numbers of library units can give rise to name clashes, either when combining software from two applications programs, or when re-using software originally developed as parts of a number of different applications programs (or from re-use libraries of different origins).

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

There are a number of possible solutions:

a) Multi-Level Program Library

This would be an LRM change. It involves a way of identifying groupings ("shelves") of library units that will be explicitly reused, and limiting the visibility of library unit names to the groupings in which they occur, except that they can be selected by being prefixed by the name of the grouping.

(The notion of 'subsystems' in the Rational development environment goes some way towards this, by limiting the use of the with statement, but does not address the problem of name clashes between units in different subsystems)

b) Provision of Tools to Resolve Name Clashes

E.g. a "with" of library units A and B, which were developed independently, fails because this would

bring into the program two different library units named C. This could be resolved by the use of tools that would edit B, and any units with-ed by B, to replace C by some other name that does not appear anywhere in the program. This tool could work by creating edited copies either of the sources, or of the compiled units.

Editing the sources means recompiling and is therefore expensive. It also introduces Configuration Management problems.

Editing the compiled units would mean changes to the LRM and/or validation/compiler policy rules, and the provision of such tools would have to be made a requirement on the supplier (and checked by additions to the validation suite). It might or might not cause additional Configuration Management problems, depending on the compiler.

Either way, reliance on such tools will be tedious for the user, and will make error messages difficult to interpret.

c) Stronger Requirements for Compile-Time Optimization

One way round the problems is for re-use libraries to consist of a few large packages, with conveniently short names.

However, to avoid frequent recompilations, it is necessary for the Ada compiler and linker to have the following characteristics:

i) Recompiling a library package specification and body to add declarations to cause no recompilation of existing separately compiled subunits of the library package body or of any units that "with" that package.

ii) Recompiling a library package specification and body to change declarations to cause no recompilation of separately compiled subunits of the library package body or of any units that "with" that package, except for such subunits or units that actually make use of the changed declaration.

iii) Re-importing a library package (i.e. where re-use libraries are distributed by copying Ada units or program libraries in compiled form) to cause no recompilations that would not have occurred under (i) and (ii) had the library package been recompiled locally.

iv) Unused subunits to be detected and omitted by the linker, to prevent the executable image being larger than necessary due to the inclusion of unused code.

This probably needs a change of emphasis in the LRM, so that in principle re-compilation of a dependent unit is only needed when a library specification is changed in a way that makes it incompatible with that dependent unit, but some compilers might do otherwise (c.f. generic bodies).

I personally feel (a) is best, and (b) the lease desirable.

CONFIGURATION CONTROL OF COMPILATIONS IN A PROJECT SUPPORT ENVIRONMENT

DATE: March 21, 1989

NAME: M.J. Pickett, from material supplied by members of Ada UK.

ADDRESS: Sema Group plc
Orion Court
Kenavon Drive
Reading
Berkshire RG1 3DQ
United Kingdom

TELEPHONE: +44 734 508961

ANSI/MIL-STD-1815A REFERENCE: 10

PROBLEM:

The construction of large systems requires careful attention to be paid to configuration management. Developers of large systems make use of tools to support configuration management which are components of their development environment. Such tools have the potential to keep track of multiple variants and versions of Ada compilation units.

The Ada standard does not define the program library in such a way as to guarantee either adequate or consistent support by compilers with regard to the demands of configuration management, and so that proper use can be made of configuration management tools.

The requirements placed on the implementation of the program library are limited to the minimum required for supporting the Configuration control of compilations in a project support environment mechanics of separate compilation with visibility of earlier compilation units. Failure either to define the program library sufficiently, or to express the requirements of separate compilation in a functional way, has resulted in compilation systems which are inadequate tools for the development of large systems, or which provide various essential facilities in a variety of ad hoc ways. Thus, recognizing that to provide no support at all for maintenance of the program library is unacceptable, and yet having no defined interface through which to work, compiler developers have been led to producing library management facilities which exhibit tight coupling between a compiler and a program library and which make integration of the program library with a proper configuration management system extremely difficult.

In consequence, the developer of a large application finds that there is a lack of standard support for

-multiple variants and versions of compilation units;
-access control;
-source code identification and derivation histories;
-change control;
-control of visibility;
-importing and exporting of compilation units;
-general reporting.

It is not seen as being necessary for the Ada language to address these requirements directly. However, it is seen that the way in which the language is defined by the current standard is deficient and has resulted in the continuing lack of standard support for the means to meet these requirements.

IMPORTANCE: ESSENTIAL

ESSENTIAL for other than small, one-off systems.

Without standardization, developers of environments will be reluctant to support more than one Ada compilation system, with a resulting restriction of choice to the user, and a consequent reduction in the justification for using Ada.

Without standardization, developers of Ada compilation systems will continue to develop their own ad hoc approaches which may result in the somewhat unsophisticated exploitation of the facilities provided by whatever support environments they employ. A less than well integrated Ada compilation system is a discouragement for prospective users of Ada.

Without efficient and comprehensive configuration management systems for Ada, developers of large systems are likely to choose other languages.

CURRENT WORKAROUNDS:

In general, developers of Ada compilation systems have recognized the inadequacy of the standard and have implemented program library facilities which both conform to the standard and go some way to meeting the real world requirement for configuration management. Each developer has adopted a different approach and provided a different collection of facilities. Therefore there is no general workaround to develop facilities which are missing, nor to integrate an Ada program library with a more general configuration management tool.

POSSIBLE SOLUTIONS:

10.4 (3) assigns responsibility for aspects of the program library to the programming environment. To ensure that environments support Ada compilation systems more consistently, a standard is required which defines a more comprehensive interface to the environment.

Steelman 12A calls for a structured library to allow entries to be associated with particular applications, projects, and users. The design of Ada neglected this. 9X is an opportunity to put this right.

There are four levels at which standardization might be applied. At the top level, the language could be extended to address the relevant issues. At a level only slightly lower, there could be a secondary standard defining services and interfaces to be provided for interaction between a program library and the programming environment. At a level lower still, these services and interfaces could be defined in an optional addendum to the standard. At the bottom level there could be as requirement for the developer of the compilation system to list which of a defined set of services and interfaces he has provided and to specify how they may be accessed.

Although it may be recognized that the program library, as currently defined, was intended only to provide the basic facilities, it can now be seen that this view was short-sighted and should not be maintained. There is an urgent need to amend the standard to express the functional requirements for separate compilation without defining the program library per se. This amendment should also address the requirement for a standard interface for configuration management systems. The most attractive approach is for the actual

interfaces for configuration management to be defined in a secondary standard which is developed in the same time frame as the primary standard.

SEPARATE COMPILATION INDEPENDENT OF A PARTICULAR LIBRARY MODEL

DATE: July 24, 1989

NAME: D J Tombs, and endorsed by Ada-UK

ADDRESS: RSRE,
St. Andrews Road
Great Malvern
Worcestershire
WR 14 3PS, UK

TELEPHONE: +44 684 895311

ANSI/MIL-STD-1815A REFERENCE: 10, especially 10.1 and 10.4

PROBLEM:

We cannot adequately configure large systems as the language now stands. There are no standard means of performing the kind of operations on library units generally considered desirable. These include:

- creating a new variant or version of a compilation unit;
- mixed language working, particularly the use of Ada units by other languages;
- access control, visibility of units to other programmers;
- change control and the general history of the system.

The inability to do these things arises out of a few loosely worded paragraphs in the LRM (in 10.1 and 10.4), which imply the existence of a single Ada program library, whose state is updated solely by the compiler. This can be an inconvenient foundation on which to build. The relationships between compilations in a project will be determined by the problem and the organization of work, and any automatic enforcement of a configuration control regime must come from a locally chosen PSE. Ada especially, as a language with large and diverse application, must have a separate compilation system which gives the greatest freedom possible in this area.

An example

This example illustrates the problems of implementing variant control using Ada libraries. Suppose an application has been decomposed into three components, A, B and C. A uses B, B uses C. Suppose the manager now conceives a different variant of A (A') which uses a different B (B'), where both variants should simultaneously benefit from changes in C (which is to be the subject of some experimentation). Now consider how this history is to be modelled in Ada.

Suppose the two variants lie in the library. To avoid a clash, one is lead to use names such as "A_V2" and "B_V2.FRED" in the text on the new variants. This is a regrettable intrusion of general concerns on the details of implementation: Ada goes to great pains to allow the programmer to use functionally significant names. Moreover, there is nothing to stop the writer of A_V2 from using B and B_V2 at once. There is no logical structure in this library which reflects the development of distinct variants: the configuration information implicit in the names is wasted on the compiler. Otherwise, if two libraries are used for the parallel variants, what happens to subsequent versions of C? If there is a copy of C in each

library, then recompilation rules applied to C do not enforce the requirement that each variant of B uses the same version. If there is just one copy, how does the compiler know in which library to look for it? Just one library is the universe as far as the compiler is concerned.

DISCUSSION

A separate compilation system must tackle issues of integrity, identification and the limitations it places on configuration control. By integrity, we mean those mechanisms whereby entities are used only in appropriate ways. By identification, we mean the problem of deciding whether names mentioned in two places refer to the same entity.

The Ada language enforces integrity through the type system and tackles identification thorough the name rules. Products of previously compiled units are introduced as Ada declarations visible in the environment of the new unit, thus subjecting the use of these products to the same type discipline as would be obtained in a monolithic compilation. Name identification within a library is achieved by creating a compiler for each library, which is the universe for that compiler, and insisting on unique names therein. The example demonstrates that this mechanism introduces a coupling between configuration and the names used inside compilations.

By so imposing a configuration system on the library, Ada limits the ability to perform configuration management appropriate to the application. For example an Ada library is the "moving front" of its constituent units. Some library units may be in an obsolete state, in that they refer (indirectly) to superseded compilations. Moreover, bodies are placed on the same footing as specifications, in that there is one for each library unit. It is thus difficult to have a library containing two versions of a program that are simultaneously consistent and runnable, as is often desired.

IMPORTANCE: ESSENTIAL

Ada was intended for use in large projects, involving many people, possibly at different centers. These are precisely the projects which will collapse if the programming support technology is inadequate.

CURRENT WORKAROUNDS:

Compiler vendors have attempted to provide configuration management facilities, for example by building sublibraries or sharing units between libraries. All such solutions are invariably messy and probably step outside the strict definition of the language - if a unit in one sublibrary is changed should the change propagate to other copies of the unit?

There is no standardization between these proprietary systems. This is not surprising in view of the lack of knowledge of the kind of facilities required at the time Ada was defined.

POSSIBLE SOLUTIONS:

We believe that the library mechanism lies entirely outside the language. This intent is also stated in the LRM [10.4(3)]. To achieve this goal it is necessary that the compiler be independent of any library; rather a PSE should contain mechanisms to create a context for a compilation and to use the result of the compilation for updating purposes.

Ada requires a separate compilation system which tackles integrity and identity, and insofar as is possible given these constraints, imposes no configuration control. Higher-level systems can be built on top of the primitives defined. This would allow the user to choose a means of configuration management appropriate

to the application.

To this end we suggest the compiler should be a pure function, having no built-in context. The compiler produces a compiled unit which is a value: there is no primitive notion of updating
The unit has a source text and a context, which is a list of other compiled units, along with Ada names by which they are to be recognized during compilation. To represent the various kinds of Ada compilation unit (specification, body, subunit, generic etc) there correspond several types of unit value. Body units correspond to just one specification unit, similarly for subunits.

A body or subunit is bound to its specification or parent, which must in consequence be part of the environment provided for its compilation. However, any number of bodies can be made for one specification, reflecting a variety of conforming implementations. Bodies do not form part of a context; the association of a specification with a body is a linking and elaboration issue.

Prior to the run-time elaboration of a unit each specification in its context must be associated with one body, thus solving the problem of identity without relying on names. In conventional terms this is linking. It is possible that unit elaboration order can be determined from the link correspondence.

The above discussion and proposal suggest that a prescribed library mechanism would not be part of Ada 9X. Rather there would be a specification for an Ada compiler and a separate statement specifying the functionality of a PSE and compiler interface, perhaps a secondary standard to the language. Such a PSE would be extensible (to allow new tools to be added and other compilers to be bound) and its use optional (to allow programming in alternative environments).

PROGRAM STRUCTURE

DATE: June 9, 1989

NAME: Judy A. Edwards

ADDRESS: General Dynamics
P.O. Box 748, MZ 1746
Fort Worth, TX 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 10

PROBLEM:

In consideration of the overall program structure of a large scale program written in Ada, several integrity questions are brought into question by the LRM definition of the various structures. Even with separate specs, the entry points are the only information available as to the realtime (even not so realtime) execution context of a unit. The contexts that are hidden because they are expressed far down into the source within the unit are as follows: representation clauses, conditional entries, delays, exception/interrupt traps, rendezvous, and deferred declarations. These execution context items are not within the first part of the header information that sets the context is a readily identifiable manner. Instead, the reader must closely peruse code (local as well as in separate compilation units). Just not all of the dependencies can be easily located.

During debug time, when overflows/contention/exceeding duty cycle and other such problems will be very difficult to track down in Ada programs. This is due to three causes: buried contexts, the excessive fragmentation of programs, and numerous implicit ways to exit a units operational context. These are very unfortunate language designs. They may not manifest themselves so much in main frame applications, but they represent serious software integrity issues for the embedded community.

IMPORTANCE:

CURRENT WORKAROUNDS:

In the programming standards: Limit the ways that "hidden" contexts can occur, provide context data in the header comments, place context controls in few places. Try to get the vendor to provide cross reference information reports from compilations or develop special tools.

POSSIBLE SOLUTIONS:

In LRM, revise the structure of a program or a unit so that all of the contexts are readily identifiable, e.g., allow USE AT to immediately follow the instance of the type declaration. Flag delays and exceptions as special entry points in the specs. (Best of all is to only allow single entry/exit points where you delay before activating some unit).

THE PROGRAM LIBRARY

DATE: August 31, 1989

NAME: Michael Sink (Canadian AWG #004)

ADDRESS: Leigh Instruments
c/o R. Leavitt
PRIOR Data Sciences
240 Michael Cowpland
Kanata, Ontario

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: 10

PROBLEM:

The program library concept as introduced in Chapter 10 of the LRM requires clarification and modification in order to allow the developer to benefit from program library created and updated by the compiler.

A specific issue of concern is related to the use of the program library to support the separate compilation rule (this includes compilation dependencies arising from a library unit's context clause.) A minor change to an existing unit can force a major recompilation, some of which may be unnecessary. The idea of separate compilation is important, and it has been introduced to "reduce compilation costs and to simplify development" (the Rationale, chapter 10) by ensuring that any change affected by this change" (LRM,10.3(5)) as obsolete (note the work "potentially"). In the case of large and complex software systems; however, this is too restrictive and can have the opposite effect.

A second issue is the general deficiency of the vendor supplied means of access to the program library. Although the program library principally supports compilation rules,it should also (at least for the sake of assisting software engineering) allow the developed to use the program library to support his software development methodology. Therefore, it is recommended that access to the program library be open to any Ada program or to tool that the developer may develop as part of his APSE.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Issue 1:
None. In the extreme case,adding a comment to a package specification could make the program library obsolete, and require the entire system to be recompiled.

Issue 2:
Current access to the program library is restricted to vendor supplied library management tools. While these can be useful, they often are insufficient and hinder configuration management.

POSSIBLE SOLUTIONS:

Issue 1:
There are instances during software development where it is necessary to make a "quick" change or modification in order to test a fix for a bug. In order to do this and avoid the possible recompilation issue, the integrity of the program library could be temporarily sacrificed. The hard separate compilation rule could be relaxed. The recompiled unit should still mark affected library units as obsolete, but still allow an executable program to be created from the suspect (invalid) library. A library status could indicate this status change. Returning the library to a valid status would be accomplished by recompiling the units marked as obsolete.

Issue 2:
One solution for program library access is to define a separate standard; however, this is not recommended.

A simpler solution which could also address issue 1, is to add to the LRM a program library package specification, which would be a language requirement in the same way that the predefined library units (STANDARD, CALENDAR, SYSTEM, etc.) are required. This way developers can use the program library to their advantage, while propriety library structures of compiler vendors would not be compromised. A sample package specification for such a program library follows.

```
package PROGRAM_LIBRARY is
-- Types
type LIBRARY_NAME is string( implementation-defined );
type LIBRARY_PASSWORD   is string( implementation-defined );
type VERSION_ID     is string( implementation-defined );
type LIBRARY_MODE       is ( GRANITE, JELLO);
type LIBRARY_STATUS     is ( VALID, SUSPECT);

type UNIT-LIST      is private;
type LIBRARY-UNIT   is private;
-- other types as needed

-- Library access services
        procedure       OPEN ( NAME : LIBRARY_NAME);
        procedure       CLOSE ( NAME : LIBRARY_NAME);
        procedure       LOCK_LIBRARY( PASSWORD : LIBRARY_PASSWORD);
        procedure       UNLOCK_LIBRARY( PASSWORD : LIBRARY_PASSWORD);
        procedure       COPY_LIBRARY( FROM : LIBRARY_NAME: : LIBRARY_NAME);
        procedure       CREATE_LIBRARY( NAME : LIBRARY_NAME);
        procedure       DELETE_LIBRARY( NAME : LIBRARY_NAME);
        procedure       WRITE_VERSION_ID( NAME : LIBRARY_NAME; ID : VERSION_ID);
        function        READ_VERSION_ID( NAME : LIBRARY_NEMT) return VERSION_ID;

        -- Library contents services
        function        LIST_OF_LIBRARY_UNITS( NAME : LIBRARY_NAME) return
                                UNIT_LIST;
        procedure       FIND_FIRST_UNIT( OF : UNIT_LIST);
        procedure       FIND_LAST_UNIT( OF : UNIT_LIST);
        function        NEXT_UNIT( OF : UNIT_LIST) return LIBRARY_UNIT;
        function        PREVIOUS_UNIT( OF : UNIT_LIST) return LIBRARY_UNIT;
        function        NUMBER_OF_UNITS( IN : UNIT_LIST) return natural;
```

```
-- and other services to extract unit information,
-- for example: teimstamp, dependencies, type, status, etc.

-- similar library services for library structure access

-- Library unit access
procedure COPY      (UNIT : UNIT_NAME;
                     FROM : LIBRARY_NAME; TO : LIBRARY_NAME);
procedure           DELETE( UNIT : UNIT_NAME; FROM : LIBRARY_NAME);

-- Library status and mode services
procedure           SET_MODE( OF : LIBRARY_NAME; TO : LIBRARY_MODE);
function            GET_MODE( OF : LIBRARY_NAME) return LIBRARY_MODE;
function            LIBRARY_STATUS ( NAME : LIBRARY_NAME) return
                    LIBRARY_STATUS;

private

-- implementation dependent

end PROGRAM_LIBRARY;
```

SUBPROGRAM REPLACEMENT

DATE: October 22, 1989

NAME: Arnold Vance

ADDRESS: Afflatus Corp.
112 Hammond Rd.
Belmont, MA 02178

TELEPHONE: (617) 489-4773
E-mail: egg@montreux.ai.mit.edu

ANSI/MIL-STD-1815A REFERENCE: 10, 6, 8.5

PROBLEM:

When defining overloaded subprograms or functions with operator designators in the visible part of a package specification, it is not possible to implement these subprograms as subunits without the introduction of extraneous subprograms in the visible part.

IMPORTANCE: IMPORTANT

There is no way to prohibit a user of the package from referring to the extraneous subprograms. Requiring that subprograms be implemented within the package body seems to defeat the subunit construct in particular and good program partitioning in general.

CURRENT WORKAROUNDS:

Here is an example of what is desired:

```
package pkg1 is
        type t is private;
        --one example for each type of subprogram mentioned above

        function "<"(l, r: t) return boolean;
        procedure op(p1: in out t);
        procedure op(p1, p2: in out t);
        ...
        end pkg1;
```

Unfortunately, there is no way to implement all of these subprograms as subunits, so the package must be modified to:

```
package pkg2 is
        type t is private;
        --these subprograms required for compilation purposes only function lt(l, r: t) return boolean;
        --end of required subprograms
```

```
function "<"(l, r: t) return boolean renames lt;
procedure op(pl: in out t);
procedure op(pl, p2: in out t);
...
end pkg2;
```

And then in the package body:

```
package pkg2 is

    function lt(l, r: t) return boolean is separate;
    procedure op(pl: in out t) is separate;
    procedure op2(pl, p2: in out t) is separate;
    ...
    end pkg2;
```

POSSIBLE SOLUTIONS:

Introduce a construct similar to renaming that allows a subprogram body to be replaced by another subprogram body. The requirements of 8.5(7) (substituting "replacing" for "renaming") should be met. In particular, a function designated by the equality operator "=" can be implemented as a subunit (because of 6.7(5), there is no way of accomplishing this currently). The suggested syntax is:

replacing_declaration::=subprogram_specification replaced by subprogram_or_entry_name;

The corresponding package body for the first package specification then becomes:

```
package body pkg1 is

    function lt(l, r: t) return boolean;
    procedure op2(pl, p2: in out t);

    function "<"(l, r: t) return boolean replaced by lt;
    procedure op(pl, p2: in out t)replaced by op2;

    function lt(l, r: t) return boolean is separate;
    procedure op(pl, p2: in out t) is separate;
    procedure op2(pl, p2: in out t) is separate;

    end pkg1;
```

SELECTION OF MACHINE DEPENDENT CODE

DATE: October 12, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 10, 13

PROBLEM:

There are no constructs in the Standard that can be used to select code based on its implementation dependencies.

IMPORTANCE: IMPORTANT

A portable program normally consists of a large collection of portable units and multiple versions of non-portable units.

CURRENT WORKAROUNDS:

Use various source management utilities to control the portable and non-portable units as separate collections.

POSSIBLE SOLUTIONS:

Add a pragma that takes as its argument a list of enumeration literals which identify the possible values of SYSTEM.SYSTEM_NAME that subsequent compilation units are targeted for. This allows construction of a compiler that picks the appropriate versions of non-portable units from a compilation that might include multiple versions of the same unit, each one targeted for a different machine or group of machines.

Note that this pragma, if accepted by the compiler, changes acceptance of units into the program library.

SECONDARY UNITS AS IMPLICIT SPECIFICATIONS

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 10.1

PROBLEM:

The current semantics of 10.1 (6) introduces a "wart" in the library concept. In connection with obsolescence rules, it leads to unnecessary recompilations. Consider the compilation sequence

```
        procedure foo is begin ...-- 1. body
end foo;
        with foo;
        package Q is ... end Q;

        procedure foo is begin ...-- 2. body
end foo;
```

Here, the rules of 10.3 (seem to) require that Q is now obsolete, although there is obviously no good reason for it. (There are ACVC checks of this ilk, despite the last sentence of 10.3(5).)

If one always required separately compiled specifications or, better, specified the semantics of compiling secondary units without previous specifications as implicitly generating such specifications (rather than serving both as library and secondary units), these problems and user inconveniences would disappear.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The notion of 10.1 (6) that a secondary unit without previously specification acts both as a library unit and secondary unit should be altered to make these implicit specifications "first-class citizens", so that compilation of such a subprogram declaration creates both a library and a secondary unit.

GLOBAL PACKAGE/PARM CONTROL

DATE: June 9, 1989

NAME: Judy A. Edwards

ADDRESS: General Dynamics
P.O. Box 748, MZ 1746
Fort Worth, TX 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 10.1.1, 10.1.2, 10.1.4, 8.3

PROBLEM:

Ada does not support an efficient way to provide a global context for all separate compilation units that are to be compiled with a set of standard context or control parameters. The code during lab checkout may have to be changed and recompiled to remove some type checking or to interface with a slightly different environment. The configuration management requirements for embedded software would require separate change paper and sell-off for each version when only different contexts are needed for the area that is being tested.

Expand the context clause to allow the compiler system parameters to establish the context for all compilation units for a program. It appears that Pragma can appear before a subunit. It also appears that a package can contain just the pragma. For embedded systems, these globally applied pragma could apply to the Main program which should guarantee that they would be applied to every subunit. In order to assure the processing, then the language should state which pre-defined pragma are meaningful when applied to the outer scope of a program, e.g., OPTIMIZE, SUPPRESS, and should further require the same definition be provided in appendix F for implementation supported pragma, READONLYMEMORY.

IMPORTANCE: IMPORTANT

Provides better system build control without impacting the configuration management for compiling with more than one set of options.

CURRENT WORKAROUNDS:

Much systems programming and special compiler/ linker tools. Several compilations created through language sensitive editors to append new pragma based upon the "build" desired.

POSSIBLE SOLUTIONS:

<<moderate change>>

In para. 10.1.1 #1 first sentence add "or within the parameterization of the compiler system".

In this manner, TEXT_IO, unchecked_conversion, pragmas, math_pack_libraries, etc. do not have to appear as the context for compiling every unit.

MACHINE CODE INSERTIONS

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 10.1.1 and 13.10

PROBLEM:

Machine code insertions are not in keeping with principles for highly reliable, maintainable, supportable code. In particular, most compiler implementations insert Hex strings to represent executable binary. What is needed is the ability to have an INLINE assembler macro expanded, frequently referred to as a built in function.

A built in function approach has the flexibility that is needed and the expressibility to support embedded applications. The vendors can easily support the capability within the context of normal INLINE procedures.

IMPORTANCE:

High for embedded systems. The system will have to interface with machine code for a variety of reasons: self-test routines for the hardware, special IO protocols and timing, built in functions, interrupt structure definition, memory mapped IO. It would be nice that not every programmer would have to insert assembler for IO and other purpose where minor built in functions can be used. Also, there's a great speed advantage for the instruction mapping to a hardware function when the compiler implementation would have difficulty in recognizing the special sequence.

Machine code insertions are unreadable, difficult to maintain, and highly compiler implementation dependent.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

A more direct approach that keeps "binary" images out of the source code is for an INLINE macro to be expanded. The macro can use specific machine code instructions by the compiler without generating a "call", but the code will appear like a "call". In the example, it is hard to tell the affect of the elaboration rules as to whether the machine code package will get expanded before a user can direct where it is to be expanded.

Also, the language definition should do the following:

1. delete machine code insertion subpara. 13.8 #2,3, and 7. Modify #1, 4, and 7 to provide for an inline macro. For example, the compiler system can be clued in as to the desired access either by the appropriate type specification and if necessary add a pragma inline reference. These two methods are as follows:

   (a) as a procedure declaration the user writes:

   ```
   TYPE MACHINE_CODE( rega, regb) returns regc:
                                          builtin machine_code;
   ```

   and to access the inline macro the user writes:

   ```
   MACHINE_CODE (REGA=>4, REGB=>5);
   ```

   or

   (b) in another form keeping with the capability already in the language

   ```
   Pragma INLINE MACHINE_CODE;
   TYPE MACHINE_CODE ( rega, regb) returns regc:
                                       builtin is separate;

    --for machine code
    --resembling a call to an assembler routine
   ```

   in accessing the inline macro the user writes the above calling sequence, where MACHINE_CODE is a macro definition that exists in binary. In this manner the code sequences have the appearance of source.

   Also, from a management point-of-view, such machine code can be isolated to a set of well-known, well documented routines for the source application to use.

THE PRIVATE PART OF A PACKAGE SHOULD HAVE ITS OWN CONTEXT CLAUSE

DATE: October 12, 1989

NAME: B. A. Wichmann (endorsed by Ada UK)

ADDRESS: National Physical Laboratory
Teddington, Middlesex
TW11 OLW. UK

TELEPHONE: +44 1 943 6076 (direct)
+44 1 977 3222 (messages)
+44 1 977 7091 (fax)
E-mail: baw@seg.npl.co.uk

ANSI/MIL-STD-1815A REFERENCE: 10.1(1), 7.4

PROBLEM:

Given the need to provide good functionality via private types, the actual implementation of the private type is likely to depend upon other facilities which should be hidden from users of the package. As it stands in Ada 83, this degree of information hiding cannot be done. Since the private part uses the same context clause as the package specification, any facilities needed for the implementation of the private type must be included in the context of the package specification. It is then difficult to determine that there is no unnecessary dependence upon the packages introduced just to implement the private types.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

The workaround is not to hide information as good design would indicate.

POSSIBLE SOLUTIONS:

Alter the syntax of private part to allow a context clause before the basic declarative items. Note that the complexities which arise in having two texts with different context clauses already arise in Ada 83 since the package specification and package bodies do not necessarily have the same context clauses.

CMDLINE

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 10.1(8)

PROBLEM:

It is difficult, in a standard manner, to get at the operating system command line arguments supplied when the program is invoked.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Look up in vendor-specific manuals the method of accessing the command line parameters and access them differently on different operating systems.

POSSIBLE SOLUTIONS:

Require that 10.1(8) be changed from:
"In any case, every implementation is required to allow, at least, main programs that are parameterless procedures, ..."
to have appended to it:

> "In all target implementations under any operating system provides the capability of passing a command line parameter or parameters to a program it starts up, the compiler shall allow the main subprogram to have at least a parameter of type IN STRING which shall contain the command line string arguments and if there are more than one, concatenated and separated by a space.
>
> In addition, in all implementations under any operating system which recognizes a system return code, the implementation shall provide the capability of defining the main subprogram as a function both with and without the string argument specified previously. If the result type is integer, this integer shall be returned to the operating system as a condition code. If boolean, true shall indicate the success code to the operating system, and false shall indicate a failure code.
>
> In addition, all implementation shall provide
>
> ```
> package SYSTEM_PARAMETERS is
>  --| In the following system parameters are numbered as:
> ```

```
    --| param O: the name of the program itself as invoked
                 from the command line.
    --| param i: where i>=1: parameter i.
    function Get_parameter_count return natural;
    --| Returns the maximum valid argument to Get_parameter.
    function Get_parameter( Number: natural ) return string;
    --| Gets system parameter whose number is specified.
    procedure Get_parameter(     Item   : out string;
                                 Last   : out natural
                                 Number : in natural );
    --| Sets Item( 1.. Last) to system parameter #Number.
    --| Sets Last to the number of items loaded into Item.
    Invalid_parameter_number : exception;
    --| Raised if the Number parameter to Get_parameter is
    --| out of range.
  end SYSTEM_PARAMETERS;
```

Implementations in embedded systems where command line parameters are not meaningful are exempt from this requirement.

COMMENT:

This feature is available in all implementations of C, so why not Ada, too?

A better implementation would be to not have package SYSTEM__PARAMETERS but to allow a parameter of type:

```
type system_parameter is array ( natural range<> ) of string;
```

Unfortunately, arrays of unconstrained strings are not permitted. (They should be.)

OVERLOADING OF COMPILATION UNIT NAMES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 10.1(10), 10.2(5), 10.2(8)

PROBLEM:

The language currently does not permit the names of compilation units to be overloaded in a given program library. The language also does not permit the names of compilation units to be operators (e.g. "="), even if they are subunits. In both cases, this forces the programmer to use a less satisfactory name for one of the compilation units, which subverts the cause of object-oriented programming (since there is often a single ideal name for both, such as PUT). These restrictions are particularly annoying because there do not appear to be any reasons for them other than unwarranted concerns about the difficulty of compiler and/or library manager implementation.

IMPORTANCE: ESSENTIAL

This revision request is motivated primarily by concerns about symmetry and aesthetics, but it also yields some practical benefit to programmers because they can now choose the best means for compilation units without interference from the compiler and/or library manager.

CURRENT WORKAROUNDS:

Use less satisfactory names for compilation units and use renaming to clean up the names where the are referenced.

POSSIBLE SOLUTIONS:

Loosen the standard so that it is allowable to both overload compilation unit names and to give compilation units the names of operators.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

ALLOW SEPARATE COMPILATION OF SUBUNIT SPEC

DATE: October 19, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 10.2

PROBLEM:

Subunits as currently defined in Ada introduce a number of undesirable situations from a software engineering point of view. One of the most painful is that if two subunits wish to make use of a common routine (rather than duplicating code), there is no way to add the common routine without recompiling the parent unit to add either a new local subprogram, or a stub for a local subprogram.

Furthermore, it is often the case that relatively independent packages are forced to be all subpackages/subunits of a single common package because they all need visibility on a single common private type. This unfortunately creates a larger-than-ideal package, and forces a user of any of the component packages to be aware of all of the other component packages, necessitating their recompilation when the spec to any component changes.

IMPORTANCE: ESSENTIAL

I believe providing better modularity is essential to practical use of Ada for large systems. Without some change, people will avoid the use of private types so that relatively independent packages may be kept separate, and will duplicate code in subunits rather than introduce a change in a parent unit.

CURRENT WORKAROUNDS:

The current workaround is to simply avoid the use of private types, or to duplicate code in multiple subunits.

POSSIBLE SOLUTIONS:

Rather than requiring the spec/stub for a subunit be within the enclosing unit, it should be separately compilable. This would make the relationship between subunits and parent units exactly analogous to the relationship between library units and package standard. To gain visibility on a separately compiled subunit spec, it would be necessary to name it in a with clause,
using expanded-name notation.

Here is some proposed syntax:

<secondary_unit> ::= <subunit_declaration>

<subunit_declaration> ::=
separate (<parent_unit_name> [body]) <unit_declaration>

<unit_declaration> ::=
<subprogram_declaration> | <package_declaration> |
<generic_declaration> | <generic_instantiation> |
<task_declaration>

<with_clause> ::= **with** <compilation_unit_name>
{,<compilation_unit_name>};

pragma ELABORATE (<compilation_unit_name> {,<compilation_unit_name>);

The new syntactic category <subunit_declaration> allows for the declaration of a new component of a package, or of a compilation unit body if the optional "body" is specified after the <parent_unit_name>.

This new component is only visible to units which explicitly mention it via a <with_clause>.

If the optional token "body" is specified, then the unit may only be referenced in a <with_clause> by another subunit of the same compilation unit.

The new component is implicitly declared within the parent unit, within the visible part if the parent unit is a package spec, or within the declarative part if the parent unit is a compilation unit body.

Such components are implicitly declared in an order consistent with dependencies implied by <with_clauses>.

If a body is provided for the new component, it is implicitly defined within the declarative part of the parent's body in an order consistent with dependencies implied by pragma Elaborate's.

Here is an example, based on the CAIS definition

```
package CAIS is
   -- Define CAIS-wide private type
   type NODE_TYPE is private;
private
   type NODE_TYPE is ...
end CAIS;

separate (CAIS)
package NODE_MANAGEMENT is
   procedure OPEN(NODE: in out NODE_TYPE; INTENT: ...);
   . . .
end NODE_MANAGEMENT;

separate (CAIS)
```

```
package ATTRIBUTE_MANAGEMENT is
   procedure SET_NODE_ATTRIBUTE(NODE: NODE_TYPE; ATTRIB: . . .);
   . . .
end ATTRIBUTE_MANAGEMENT;

separate (CAIS body)
package UTILITIES is
   -- A package local to the CAIS body
   procedure PARSE_PATH(...);
   . . .
end UTILITIES;

with CAIS.NODE_MANAGEMENT;
with CAIS.UTILITIES;
separate (CAIS)
package body ATTRIBUTE_MANAGEMENT is
   -- May call routines within CAIS.NODE_MANAGEMENT (e.g. OPEN)      -- May also call routines
within CAIS.UTILITIES
   . . .
end ATTRIBUTE_MANAGEMENT;

with CAIS;
with CAIS.ATTRIBUTE_MANAGEMENT;
procedure SET_PURPOSE(NODE: CAIS.NODE_TYPE; VALUE: STRING) is
   -- May call routines within CAIS.ATTRIBUTE_MANAGEMENT
begin
   CAIS.ATTRIBUTE_MANAGEMENT.SET_ATTRIBUTE(NODE, "PURPOSE", VALUE);
end MY_TOOL;
```

SUBUNITS IN THE DECLARATIVE PART OF A COMPILATION UNIT

DATE: June 23, 1989

NAME: Jeffrey R. Carter

ADDRESS: Martin Marietta Astronautics Group
MS L0330
P.O. Box 179
Denver, CO 80201

TELEPHONE: (303) 971-4850
(303) 971-6817

ANSI/MIL-STD-1815A REFERENCE: 10.2(3)

PROBLEM:

Subunits are only allowed for bodies which occur in the immediate declarative part of a compilation unit.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Make the declarative part containing the body the immediate declarative part of a compilation unit.

POSSIBLE SOLUTIONS:

The restriction on subunits being in the immediate declarative part of a compilation unit effectively means at the highest level of nesting of the compilation unit. This prohibits constructs such as:

```
procedure Main is
  -- declarations to allow user input
begin -- Main
  -- obtain user input
  program : declare
    -- declarations based on values of user input
  procedure Help is separate;
begin -- program
  -- the real body of the program goes here and uses help
  end program;
end Main;
```

The restriction on the nesting level at which a subunit is declared should be eliminated. Subunits should be legal at any nesting level, since the body's parent's full expanded name is required to compile the body, and this provides the necessary information.

This change would require the reference to the "parent unit" in 10.2(5)ff to be changed to the parent

declarative region.

BODY STUBS

DATE: June 7, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 10.2(3)

PROBLEM:

Body stubs can only be used at the first level (compilation units).

Is there a valid explanation for disallowing body stubs that are nested with a nested package? Compiler execution time and program library disk space penalties is not a valid reason since this penalty will only be paid by those who use the feature.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Do not use stubs, or use two stubs.

POSSIBLE SOLUTIONS:

Drop 10.02(03), compiler vendors that implemented their compiler based on Software Engineering principles won't suffer and all people that learn and use Ada will benefit.

UNIQUE PATH NAME FOR SUBUNITS

DATE: October 13, 1989

NAME: David A. Smith, on behalf of SIGAda Ada Language Issues Working Group

ADDRESS: Hughes Aircraft Company, A1715
16800 E. Centre Tech Parkway
Aurora, CO 80011

TELEPHONE: (303) 344-6175
E-mail: dasmith @ ajpo.sei.cmu.edu or
E-mail: smith @ cel860.hac.com

ANSI/MIL-STD-1815A REFERENCE: 10.2 (5) sentence 3

PROBLEM:

The following comment represents discussion that took place at the January 1987 and December 1987 meetings of the Ada Language Issues Working Group (ALIWG). Currently a subunit must have a unique simple name among all the subunits of a given ancestor library unit. It is proposed that this restriction be relaxed so that uniqueness is required only for the full expanded name of the subunit.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

It is recognized that difficulties caused by the present rule can be worked around using renaming.

POSSIBLE SOLUTIONS:

It is proposed that the following (presently illegal) program be made legal:

```
package body P is
  procedure P1 is separate;
  procedure P2 is separate;
end P;

separate (P)
  procedure P1 is
  procedure G is separate;
begin
  null;
end P1;

separate(P)
procedure P2 is
        procedure G is separate;
begin
        null;
```

```
end P2;
```

The vote was unanimous in favor of this proposal.

SOME SUBUNITS CANNOT HAVE ANCESTOR UNITS

DATE: June 5, 1989

NAME: David F. Papay

ADDRESS: GTE Government Systems Corp.
P.O. Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 10.2(5)

PROBLEM:

10.2(5) states:

> "Each subunit mentions the name of its parent unit, that is, the compilation unit where the corresponding body stub is given. If the parent unit is a library unit, it is called the ancestor library unit."

If a body stub is declared in a package body that is a library unit body, then the package body is the parent unit for the corresponding subunit. However, the package body is not (by definition) a library unit. Hence, the subunit does not have an ancestor unit.

(See also related Ada Commentary AI-00035.)

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Reword this paragraph so that the intent of the standard is achieved.

If the parent unit is a library unit, it is called the ancestor unit; if the parent unit is a library unit body, then the corresponding library unit is called the ancestor unit."

LIBRARY UNIT ELABORATION

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 10.3, 10.5

PROBLEM:

An elaboration order based on the partial ordering defined in 10.3 does not guarantee a "good" elaboration order. Because of this, pragma Elaborate is used routinely instead of in special situations (when dealing with large software systems).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None - no support provided.

POSSIBLE SOLUTIONS:

Require the compiler/library manager to determine an elaboration order which will not result in Program-Error at run-time. The exceptions are circularities or incorrect orders induced from use of pragma Elaborate. This pragma should not be routinely used - only in special situations. In the event that no legal ordering is found, the user should be notified of this fact by the library manager.

OBSOLETE OPTIONAL BODIES

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 10.3

PROBLEM:

If optional bodies become obsolete then there is a distinct danger that linkage of the program will yield an incorrectly executing Ada program. Ada implementors should have the option of diagnosing this situation with an error message, aborting the link (or execution) attempt, and requiring that the obsolete body be explicitly deleted from the library or else recompiled.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The current interpretation of 10.3(5), requiring successful linkage of an Ada main program in the presence of obsolete optional bodies should be reversed. It should, at least, be made implementation-dependent whether or not an Ada system rejects such linkage (or execution).

SEPARATE COMPILATION OF GENERIC BODIES

DATE: October 15, 1989

NAME: Tony Orme - ATM Computer
Chris Mayers - Torus Systems
Ian Mussbaum - Dowty-Sema
(Joint authors of a forthcoming book on Reusable Software.)

ADDRESS: c/o Tony Orme
ATM Computer,
AEG (UK) Ltd.
Engineering Division,
Erksdale Road,
Winnersh,
Wokingham,
Berkshire, RG11 5PF,
United Kingdom.

TELEPHONE:

Country Code	44
Direct Line	734 441419
Via Switchboard	734 698334 ex 124
FAX	734 441397

ANSI/MIL-STD-1815A REFERENCE: 10.3 (9)

PROBLEM:

The ALRM permits some implementations to require that a generic body must be compiled in the same compilation unit as its specification. It is particularly awkward, in terms of the knockon effects, to have to recompile generic specs when the bodies are changed. The common detour (see Current Workarounds) causes extra work and is detrimental to the understandability of the code.

Furthermore, since some implementations take advantage of the above let-out clause while others do not, there is a detrimental effect on the portability of Ada.

IMPORTANCE: IMPORTANT

Effect on Current Applications:

The proposed solution will have no impact on current applications.

Consequences of Non-Implementation:

Generics will continue to be an awkward feature to implement. Non-implementation will hit hardest those who make substantial use of generics, such as layered types by followers of the Object-Oriented paradigm.

The principal impacts will be on:

1. Reusability

2. Readability
3. Use of Ada
4. Take-up of Ada (by those contemplating layering of types, etc)
5. Portability

CURRENT WORKAROUNDS:

There is a detour, which is to declare a subprogram or package, within the generic subprogram or package, to actually implement the functionality (which incidentally shows that prohibition of separate compilation for generic bodies cannot be of great assistance to compiler writers).

The principal objection to use of the detour is that it creates several extra, otherwise unnecessary entities, which reduce the readability of the code. The principal effect of this "feature is thus in opposition to the declared aims of the Ada language.

A second objection to use of the Detour is that, strictly speaking, it is non-portable because 10.3 (9) actually permits an implementation to require also that subunits of a generic unit be part of the same compilation.

POSSIBLE SOLUTIONS:

The Ada language should define separate compilation of generic bodies to be a mandatory part of the language, rather than optional (and possibly deprecated - at least one compiler vendor claims that it is now a validation requirement not to provide it). Restoration of the facility, or its provision where not provided before, is a minor change for compiler writers.

The ideal solution would therefore be to delete paragraph 10.3(9) and amend the last sentence of 10.3 (6) to suit. If compiler writers should protest that the change has too many repercussions, then notice should be given that it will be implemented at the next Ada revision.

THE TIGHT COUPLING BETWEEN COMPILER AND LIBRARY MANAGER

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 10.4

PROBLEM:

Currently, there is very tight coupling between an Ada compiler and its library manager, so that it is generally impossible to integrate an Ada compiler properly with a CM system or IPSE, so that individual units can be controlled, especially in a distributed development environment. The interface between the Ada compiler and library manager should be standardized in such a way that an entity-relational database could be used to provide an alternative library manager (as in ALS and CAIS), with compiled units easily shared and replicated.

IMPORTANCE: ESSENTIAL

CM of Ada projects may be more expensive than that of non-Ada projects.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Although alternatives to an entity-relational model might be considered, it is essential to keep all the information (other than inter-unit dependencies) about a compilation unit in one place, so that it can be stored, copied or communicated as a unit.

Typically, an Ada program library consists of a collection of files and directories. Some at least of these files are in formats for which documentation is not provided, and the information about an individual unit can only be partially accessed using the library management tools supplied with the compiler.

It is usually possible to use multiple Ada program libraries in order to hold multiple versions of a unit, under different levels of access protection, but it is difficult to exploit these fully within a more general project support environment because of the following sorts of problems:

a) Use of a large number of Ada program libraries can be expensive.

b) The available information about the units is not complete (e.g. the identity of the source file from which a unit was compiled), and is obtained via interactive commands rather than procedural

interfaces.

c) There are often restrictions on copying units or libraries, and therefore it can be difficult or impossible to make a copy of a unit in a safe place.

d) What facilities there are differ from one compiler to another so much that an Ada workbench for an IPSE has to be written from scratch for each compiler, and the provision of a useful, compiler-independent workbench (e.g. for a project using a variety of compilers for multiple targets and host testing) is almost impossible.

e) An Ada workbench is needlessly difficult to implement, and slow in use, because of the indirect way in which the operations on the program libraries are performed.

To describe a way to get round these problems, it is easiest to think in terms of a library management package, used by the Ada compiler, linker and library management tools for all their operations on program libraries. The requirements can then be stated as follows:

1) The library management package shall provide, in a standard form which is the same for all compilers (e.g. pre-defined in the LRM), the operations required by an Ada compiler to manage a single program library, plus operations to perform simple queries (e.g. list all units). Additional operations shall be provided in each implementation to control which view of the underlying data constitutes the library seen by the compiler.

2) The form of these operations, and the types upon which they operate, shall be such that it shall be possible easily to implement the package with all the information about a particular unit held in:

   i) A "file", whose structure is known to the compiler but not to the library manager. If this "file" is more complex than a simple operating system file, the compiler vendor must provide routines to manipulate it (create, copy, move etc.) as a single entity (including any updates to allow for changes of position within the file store). Such "files" must not contain any references to other files.

   +

   ii) Standard (compiler independent) variables such as creation date, unit name, unit type etc.

   +

   iii) Standard (compiler independent) types of references to other units (to express compilation dependencies).

3) The interface to the library management package shall permit recording the identify of the source file from which a unit was compiled.

4) The library manager package implementation for a particular compiler shall be supplied in such a form that Ada programs may be written to use it.

5) The compiler, linker, etc., shall be supplied in such a form that a customer-written body for the library manager package may be used.

Requirements (1) to (4) are needed to be able to write a standard Ada workbench which can be interfaced to an IPSE database. (5) makes it possible to used IPSE database (or a Configuration Management database) to provide the program library directly.

Note that the interface to the library management package must be capable of supporting more sophisticated compilers than those currently available, e.g. if B is dependent on A, the decision as to whether a recompilation of A causes B to become obsolete must by left to the compiler.

MEANING OF A SINGLE COMPILATION

DATE: August 27, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne, IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 10.4(1) and 10(1)

PROBLEM:

In 10.1(1), the LRM states that "Each compilation is a succession of compilation units." However, in 10.4(1) when attempting to define a program library, the LRM states, "Compilers are required to enforce the language rules in the same manner for a program consisting of several compilation units (and subunits) as for a program submitted as a single compilation." By the previous definition of compilation, there is no difference between a program consisting of several compilation units and program submitted as a single compilation.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: N/A

POSSIBLE SOLUTIONS:

Change the wording of 10.4(1) so that it reads, "Compilers are required to enforce the language rules in the same manner for a program submitted as several compilations as for a program submitted as a single compilation." This is presumably what was intended.

TRANSITIVE ELABORATION

DATE: July 21, 1989

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of Ada UK

ADDRESS: IPSYS plc,
Marlborough Court,
Pickford Street,
Macclesfield,
Cheshire SK11 6JD
United Kingdom

TELEPHONE: +44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 10.5

PROBLEM:

In order to achieve satisfactory elaboration of a library unit body it is necessary to determine which library unit bodies that body needs to be elaborated, and so on in a transitive manner.

For example, the pragma ELABORATE in:

```
with TEXT_IO;
pragma ELABORATE (TEXT_IO);
package body P is
begin
  TEXT_IO.PUT ("Elaborating P");
end;
```

may not be sufficient to achieve the required effect. The implementation of TEXT_IO.PUT may in turn require facilities provided by other packages. The resulting set of library units may be difficult to determine in compilation environments that impose access restrictions. Also the required elaboration will depend on the particular implementation of the library unit.

IMPORTANCE: IMPORTANT

Particularly within Ada environments.

CURRENT WORKAROUNDS:

The workaround is to determine the relevant units and provide the necessary context clauses and pragmas, e.g.:

```
with TEXT_IO, COMMON_IO, SYSTEM_IO; -- perhaps
pragma ELABORATE (TEXT_IO, COMMON_IO, SYSTEM_IO):
package body P is
begin
```

```
    TEXT_IO.PUT ("Elaborating P");
end;
```

(Note that the transitively determined units also need to be included in the context clause)

The more usual workaround is not to use the library unit elaboration mechanism at all, but to use explicit initialization procedures. This also enables the user to determine the precise order of initialization of packages.

POSSIBLE SOLUTIONS:

The proposed solution is to make the effect of pragma ELABORATE transitive. Thus, only the required library unit body would need to be specified in the pragma and hence in the context clause.

LINKAGE OPTIMIZATION

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 10.5

PROBLEM:

Multiple main programs are allowed in Ada. However, the elaboration rules are such that a compiler may fail validation if it does not elaborate every item in order in every library. The system should allow for an elaboration to occur just once and for the elaboration to take on the meaning of "access to the information". Managing multiple main programs is not part of the standard. There are few words that lead the implementor to support shared runtimes and utilities, pre-elaborated data, shared interfaces, common interrupts. A control superstructure, like an executive, has to be invented that may even duplicate much of the vendors provided runtimes for managing multiple programs. Sometimes the user has to experiment with the compilation system to determine what parts will have to be modified either in the runtime support or in the linker tools to provide efficient support.

IMPORTANCE: IMPORTANT

Embedded systems are heavily multiprogrammed to provide the time/ event dependent processing in an even way. The algorithms are too long to wait until one completes. Tasking synchronization through rendezvous adds great inefficiencies. The interleaved algorithms are easily handled outside the scope of the language with very simple control structures managing the realtime clock. Ada tasking doesn't come close to supporting the realtime needs. Also, multiple programs, or CSCIs in DoD Std 2167 terms, are envisioned in next generation systems where two programs may not even be from the same set of developers.

CURRENT WORKAROUNDS:

Much unique special handling provided and possible changes to compiler vendor's systems structures and runtimes. Plus, the user may have to have redundant code and revert to assembly language to bypass Ada's lack of definition.

POSSIBLE SOLUTIONS:

1. allow the compilation system to move some elaborations to external status to avoid expanding code in a redundant and duplicate fashion.

2. specifically state that items which are not referenced are not to be elaborated

3. allow "simple" addresses to be used as access type results. The user still should not be taking

advantage of the fact that an address is a known value, but should know and control whether there is an intermediate structure or object between the access variable and the object that is referenced.

4. allow the compiler system to delete unreachable and empty subprograms

5. allow for an external environment to manage the key machine assets so that multiple mains are allowed

6. expand notion of SHARED to extend to other constructs, e.g., interrupts, utilities, runtimes, from other libraries.

REDUCING THE NEED FOR PRAGMA ELABORATE

DATE: October 11, 1989

NAME: Mats Weber
Endorsed by Ada-Europe, number AE-025,
originator : Stef Van Vlierberghe

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: +41 21 693 42 43
E-mail: madmats@elcit.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 10.5

PROBLEM:

Pragma Elaborate creates many portability problems because the language rules do not enforce the order of elaboration of library units or the presence of pragma Elaborate where it is necessary.

In many cases, Ada 83 requires a pragma Elaborate where the compilation system could figure out the dependencies by itself.

Examples:

```
1.)-------------------------------
package A is
        function F return Integer;
end A;
with A; --In this case it is obvious that A's body needs to be elaborated here.  Ada 83 requires
        --pragma Elaborate (A).

package B is
        X : Integer := A.F;
end B;

package body A is ...
1.)-------------------------------
2.)-------------------------------
generic
package A is
end A;

with A; --in this case it is obvious that A's body needs to be elaborated here.  Ada 83 requires
        --pragma Elaborate (A).
```

```
package B is
        package Instance_Of_A is new A;
end B;

package body A is ...
2.)-------------------------------
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use pragma Elaborate and programming tools that detect dependence on the elaboration order.

POSSIBLE SOLUTIONS:

Suppress pragma Elaborate and construct a set of rules that determines a partial ordering relation on the library units AND secondary units, using the rules in LRM 3.9(5-8) and avoiding Program_Error in such cases.

Programs that contain circularities in this relation should be illegal.

NEED FOR A PRAGMA ELABORATE

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 10.5(3)

PROBLEM:

Need a pragma elaborate.

Pragma ELABORATE is difficult to use and forgetting it introduces runtime errors.

Following example describes an extremely annoying case:

```
package P1 is procedure O; end P1;
package P2 is procedure O; end P2;
package P3 is procedure O; end P3;

package body P1 is
        procedure O is begin null; end O;
end P1;

with P1;

package body P2 is
        procedure O is begin P1.O; end O;
end P2;

with P2; pragma ELABORATE (P2);
package body P3 is
        procedure O is begin null; end O;
begin
        P2.O;
end P3;
```

Since P3 calls P2.O during its elaboration, it needs to include pragma ELABORATE after its context clause, but this is not enough, a compiler still may defer elaboration of the body P1 until after that of P3, and as a result PROGRAM_ERROR could be raised during elaboration of P3's body.

Clearly pragma ELABORATE breaks the basic principles of modularity: P2 needs to include a pragma ELABORATE (P1) depending on his users needing calls to O during elaboration or not...

Note that when using the simplified example above, it is very unlikely that a compiler would not be clever enough to find a proper elaboration order, (a simple rule, keep body elaboration as close to the specification elaboration as possible will do), but the language admits it and at least some compilers expose the problem in more complex examples (involving nested generics, tasks, etc...).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Programmer discipline.

POSSIBLE SOLUTIONS:

Why not implying pragma elaborate for all packages that contain declarations which are used during elaboration of a certain package, such that one can no longer forget this pragma for most "clean" packages.

The preliminary formal Ada definition (Nov. 1980) contained a section that said:

> The order of elaboration of library units that are package bodies must also be consistent with any dependence relations resulting from the actions performed during the elaboration of these bodies. Thus if a subprogram defined in a given package is called during the elaboration of a body of another package (...), the body of the given package must be elaborated first.

The fact that this was changed by 1983 is probably caused by people who did not want to prevent "tricky" elaboration.

The problem with elaboration order is that there is a subtle difference between "is called" and "could be called" during elaboration. "could be called" is easy to detect, but "is called" may well be impossible to detect.

If one considers an example of a package that would not work in this context, e.g. a package named TRICKY_ELABORATION that seems to need elaboration of another package named SEEMS_TO_NEED_ELABORATION, but yet does not contain a pragma ELABORATE for the latter:

```
with SEEMS_TO_NEED_ELABORATION;
package body TRICKY_ELABORATION
        procedure USED_DURING_ELABORATION is
        begin
                if SOME_CONDITION
                then SEEMS_TO_NEED_ELABORATION.DO_SOMETHING;
                end if;
        ...
        end
begin
        USED_DURING_ELABORATION;
end TRICKY_ELABORATION;
```

This tricky package presumably works because the programmer knows that SOME_CONDITION will only be true after elaboration.

Wouldn't it be safer to require a specific pragma that tells the compiler NOT to bother about the apparent elaboration dependency when a programmer was consciously avoiding the need for elaboration? As a result "tricky" programming would be clearly flagged.

In this case the tricky package would need a pragma NOT_ELABORATE:

```
with SEEMS_TO_NEED_ELABORATION;
pragma NOT_ELABORATE (SEEMS_TO_NEED_ELABORATION);
-- SOME_CONDITION is always FALSE during elaboration because...
package body TRICKY_ELABORATION
        procedure USED_DURING_ELABORATION is
        begin
                if SOME_CONDITION
                then SEEMS_TO_NEED_ELABORATION.DO_SOMETHING;
                end if;
        ...
        end
begin
        USED_DURING_ELABORATION;
end TRICKY_ELABORATION;
```

Upward compatibility with the former Ada version is no problem at all if one generates a pragma NOT_ELABORATE for each with clause that did not have a pragma NOT_ELABORATE in the former Ada version. But for most projects, and certainly the clean ones, dropping all pragma ELABORATE's will be just fine.

A big advantage of this approach is that projects that do not use pragma NOT_ELABORATE are guaranteed after binding to always elaborate at runtime, something which is never guaranteed when using Ada83.

A last, minor problem with the suggested solution, concerning implementation only, is the work involved during binding. The binder needs to have access to :

* all compilation units
* all subprograms
* the "unit_X contains subprogram_Y" relationship
* the "subprogram_X might_call subprogram_Y" relationship
* the "elaboration_of unit_X might_call subprogram_Y" relationship

The two latter relationships can be transitively combined to "elaboration_of unit_X might_indirectly_call subprogram_Y" which, in combination with "unit_U contains subprogram_Y" gives the relation "elaboration_of_unit_X should_be_preceded_by_elaboration_of_unit_U", which should be respected by the actual elaboration order, unless specifically flagged by pragma NOT_ELABORATE.

High quality compilers that do not want to spend too much time on binding could easily implement optimizations to overcome this performance problem. In most cases the problem can be simplified to relations:

* the "unit_X contains_possible_calls_subprograms_in unit_Y" relation, which is a subset of the "with"relation and hence very cheap to maintain>

* the "elaboration_of unit_X might_call_a_subprogram_ contained_in unit_Y" relation

If elaboration order would be first determined by the second problem statement and only replaced by the first problem statements where loops occur, binding time could not be significantly increased by the different elaboration semantics.

SOLVE THE "ELABORATION ORDER" PROBLEM PROPERLY

DATE: September 15, 1989

NAME: Kit Lester
(on behalf of the Ada-Europe 9X group material supplied by Stef Van Vlierberghe)

ADDRESS: Portsmouth Polytechnic
115 Frogmore Lane
Lovedean,
Hampshire PO8 9RD
England

TELEPHONE: +44-705-598943 after 1pm EST
or E-mail to CLESTER @ AJPO.SEI.CMU.EDU
or to LESTERC @ CSOVAX.PORTSMOUTH.AC.UK
or to C_LESTER @ ARE-PN.MOD.UK

ANSI/MIL-STD-1815A REFERENCE: 10.5(3-5)

PROBLEM:

The "elaboration order" problem for bodies was observed relatively late in the Language Design process for Ada-83, with the result that there wasn't the time to give the problem the full study it demands.

As a consequence, pragma ELABORATE was introduced as a stop-gap solution with (to the best of my recollection) the intent that the issue be more fully treated before the first revision.

Since then it has become clear that the existing pragma has deficiencies: for example:

- transitivity is apparently needed: see change request 044 (by Tucker Taft of Intermetrics).

- The Ada-83 Standard says that if there is a circularity in the elaboration order required by "with"s and pragmas ELABORATE, then "the program is illegal" [10.3(5)]. That is hostile to the team responsible for a program including many units, when the compiler suddenly starts reporting such an illegality.

  In particular most "implementations" consist of a compiler and a linker, and the illegality would be reported "at link time": if many units are involved, the linker is likely to give a message which is unhelpful in identifying the cause of the problem.

- There are cases in which it is very difficult to decide whether a pragma ELABORATE is needed. Consider the following example (due to Stef van Vlierberghe):

```
with SEEMS_TO_NEED_ELABORATION;
package body TRICKY_ELABORATION is
        procedure USED_DURING_ELABORATION is
                if SOME_CONDITION
                then SEEMS_TO_NEED_ELABORATION.DO_SOMETHING;
                end if;
```

```
        ...
    end     USED DURING ELABORATION
begin --    of TRICKY_ELABORATION's initialization part
            USED DURING ELABORATION
end         TRICKY_ELABORATION;
```

TRICKY_ELABORATION needs a pragma ELABORATE(SEEMS_TO_NEED_ELABORATION); - or does it? If SOME_CONDITION is always FALSE at the time of elaborating TRICK_ELABORATION, then the pragma is not needed, and may even cause unintended circularities if it is thoughtlessly given. [Note that this case is, in general, undecidable.]

And other pathologies

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Either

- **pragma** ELABORATE each body, for all other bodies it pre-requires... or
- complex analysis of what does need pragma ELABORATE, and for what, with the results of the analysis liable to change as a consequence of apparently irrelevant changes elsewhere...
  or
- superhuman programmer discipline.

  None of them really viable.

POSSIBLE SOLUTIONS:

The known pathologies could be given piecemeal treatment: but that makes the language ever more messy and fails to protect us from further pathologies not observed in time to be treated before the 9X standard is finalized.

So the need is for a study of the problem.

The original problem (the "Brosgol anomaly") was noticed late because there were no compilers, hence no real programs running into the problem of elaboration-order of bodies. There are now compilers and many real programs: hence a lot of experience as to what is needed (but probably a significant difficulty in collecting that experience!) If we can fully understand the problem, we have a chance of a solution, rather than a patch (i.e., rather than pragma ELABORATE) However, if such a solution is found, then pragma ELABORATE would need to be retained alongside the solution in the 9X standards a depreciated feature" (i.e. present for upward compatibility from Ada-83, but with the advice that (1) it should not be used in new programs and (2) it would be removed in Ada-200X and so existing users of the pragma should aim to have removed it from their programs before Ada-200X).

RELAX REQUIREMENTS FOR ELABORATE PRAGMAS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 10.5(4)

PROBLEM:

It would be better if an improper ELABORATE pragma were rejected at compile time, rather than being ignored as required by 2.8(9). It seems that using a pragma to specify this kind of elaboration order dependency was a mistake; a different mechanism should have been used.

If, however, improper ELABORATE pragmas are going to be ignored, then it is important that the rules for determining the propriety of an ELABORATE pragma be as intuitive as possible. There are cases that arise commonly in practice of ELABORATE pragmas that appear to be correct but, for obscure reasons, are not. These restrictions should be relaxed.

In a context clause, it should be legal to intersperse with clauses and ELABORATE pragmas. For example, the ELABORATE pragmas in the following example should be accepted as semantically correct:

```
with Foo;
pragma Elaborate (Foo);

with Bar;
pragma Elaborate (Bar);

procedure Bax;
```

If one wishes to add an ELABORATE pragma to the context clause of a secondary unit, and if the object of the pragma has already been mentioned in a with clause of an ancestor unit of the secondary unit (or in the context clause of the corresponding library unit), then the secondary unit should not have to repeat the with clause in its own context clause. For example, the ELABORATE pragmas in the following example should be accepted as semantically correct:

```
with P1;
with P2;
package Foo is
end Foo;

--------

with P3;
```

```
pragma Elaborate (P1);
package body Foo is
        procedure Sub is separate;
end Foo;

--------

pragma Elaborate (P2);
pragma Elaborate (P3);
separate (Foo);
procedure Sub is
begin
        null;
end;
```

Finally, the text ", and this library unit must have a library unit body" should be deleted from 10.5(4) and the next sentence should be revised to read "Such a pragma specifies that the library unit body (if any) must be elaborated before the compilation unit". Was it really the intent of the language designers to require a link-time check that bodies are present for all library units mentioned in ELABORATE pragmas?

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Add redundant with clauses.

POSSIBLE SOLUTIONS:

CONDITIONAL COMPILATION MUST BE AVAILABLE FOR ALL LANGUAGE FEATURES

DATE: September 15, 1989

NAME: Randall Brukardt

ADDRESS: R.R. Software, Inc.
P.O. Box 1512
Madison, WI 53704

TELEPHONE: (608) 244-6436

ANSI/MIL-STD-1815A REFERENCE: 10.6

PROBLEM:

Section 10.6 says that IF statements should be used to effect conditional compilation. However, an IF statement cannot appear in directly in a declarative part. Furthermore, many pragmas are not affected by the context in which they appear.

Conditional compilation is used for many reasons. One of the most common is to support debugging code that is not present in a production version of the program. Since debugging code may include extra subprograms, tasks, and objects, it is important to be able conditionally compile these entities. Debugging code may also require library units that are otherwise unneeded. Therefore, WITH clauses also need to be conditionally compiled. Finally, pragmas also need to be conditionally compiled -- for example, to turn optimization on or off, or to suppress checking in critical areas.

IMPORTANCE: IMPORTANT

Unless this problem is addressed, users will have to continue using the ugly workarounds given below.

CURRENT WORKAROUNDS:

The code which needs to be added and removed can be commented in and out by hand. This is an error-prone and time-consuming process. Additionally, the need to make source modifications causes a configuration management nightmare. Source code may need to be changed often, simply to reflect the changing debugging needs of the programmers. Such changes can be hard to track, and can obscure the real changes being made to the code.

Alternatively, a custom-written code preprocessor can be written. This has the effect of decreasing portability and reusability, since the preprocessor will be necessary to debug or enhance the code.

A partial workaround is to use IF statements with a static condition and with block statements nested therein. This allows conditional compilation of some items, but not pragmas or with clauses. Additionally, it can lead to awkward code structuring, especially if entire procedures or tasks need to be conditionally compiled.

POSSIBLE SOLUTIONS:

The simplest solution is to define a lexical element which is treated either as a space or as the start of a comment, depending on the state of a compiler switch or a pragma. This solution does not work very well with the OPTIMIZE or SUPPRESS pragmas, since these may need to be supplied only when debugging is off. One could solve this problem by using two lexical elements, both of which work as described above, but which are treated in opposite ways. For example, if one used the lexical element "@" to denote debugging code and "~@" to denote code used only in a production version, one might write code like the following:

```
@   with VERY_SLOW_CONSISTENCY_CHECK, INTERNAL_ERROR;

procedure SAMPLE is

@       pragma OPTIMIZE (SPACE); -- Optimize for space when debugging,
@           -- since there is a lot of debugging code.
~@      pragma OPTIMIZE (TIME); -- Optimize for time in production
~@          -- versions, since this procedure is time-critical.
~@      pragma SUPPRESS (RANGE_CHECK); -- Also, in production versions,
~@      pragma SUPPRESS (DISCRIMINANT_CHECK); -- we cannot afford
~@      pragma SUPPRESS (INDEX_CHECK); -- the most expensive checks.
    begin
@       if not VERY_SLOW_CONSISTENCY_CHECK then
@           INTERNAL_ERROR ("Inconsistency in procedure SAMPLE");
@       end if;
        ...
```

A more general solution would be to provide something on the lines of the C preprocessor, or the conditional assembly found in most assemblers. This may require additional syntax and keywords, and would introduce additional complexity into the language.

OPTIMIZATION OF CONSTANT GENERATING FUNCTIONS

DATE: October 12, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 10.6

PROBLEM:

It is very difficult for a compiler to recognize that expressions like SINE(10.0) are constant.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Calculate the expression by hand and put it into the code.

POSSIBLE SOLUTIONS:

Allow a pragma that identifies constant generating functions.

Note that this allows specification of very complex initialization functions such as one that reads in a database. The compiler should limit its acceptance in a reasonable fashion.

OPTIMIZATION NEEDS SHARPER DEFINITION

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 10.6

PROBLEM:

In most respects Ada is designed to be very reliable and predictable. But, when it comes to optimizations...

A better definition of optimization is needed so that numerical algorithms can be shown to be correct. There has to be a reasonable balance between predictability and reliability of code verses speed of execution. Some of the problems that must be specified occur when the arithmetic unit has greater precision than storage and some, but not all, results are held in the arithmetic units register stack.

```
A := B;
if A > B then -- might be true
```

A specific example related to optimization:

In order to maintain full range of computation in complex arithmetic, the general technique is to do the expansion of complex cartesian representation and use component by component formulas using real arithmetic. e.g.

(a+ib)/(c+id) = ((ac+bd)/(cc+dd)+i(bc+ad)/(cc+dd))

if the components are on the order 'LAST or 'SMALL this equation could respectively underflow or overflow. The transformation for |c| >= |d| is to divide through by c squared. By properly grouping terms, either overflow or underflow, but not both,
can be eliminated.

```
(a+ib)/(c+id)= (((a/c)+(b/c)(d/c))/(1+(d/c)(d/c))+
                i((b/c)-(a/c)(d/c))/(1+(d/c)(d/c)))
```

note several things:
1 >= 1+(d/c)(d/c) >= 2 because |c| >= |d|

There are only three terms (a/c) (b/c) and (d/c) this should be computable with five divides, three multiplies and two adds and one subtract operations possibly without an intermediate store operation.

IMPORTANCE: ESSENTIAL

To a few application where life is dependent on predictable results form a computer program.

CURRENT WORKAROUNDS:

Become non portable. Study each compiler (some times each version) to see what optimizations happen. Some times use optimization breakers to get reliable numerical results.

POSSIBLE SOLUTIONS:

Enough specification to get predictable results but no so much to prevent optimization that is needed. Generally this applies to floating point computations.

For additional references to Section 10. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

NUMBER	TITLE	PAGE
0092	FINALIZATION	3-7
0117	PRE-ELABORATION	3-2
0268	SEPARATE SPECIFICATIONS AND BODIES	6-26
0365	IMPLEMENATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (I)	3-121
0432	IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (II)	3-124
0665	DISTRIBUTED/PARALLEL SYSTEMS	9-96
0757	DEFINITIONS FOR PROGRAM UNIT AND COMPILATION UNIT	15-37

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 11. EXCEPTIONS

INCLUDE "WHEN" IN RAISE STATEMENT SYNTAX

DATE: July 9, 1989

NAME: William Thomas Wolfe

ADDRESS: E-mail: wtwolfe@hubcap.clemson.edu
Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847

ANSI/MIL-STD-1815A REFERENCE: 11.3.2, 5.7.2

PROBLEM:

The raise statement is quite similar to the exit statement in that they both serve to transfer the flow of control, almost always on the basis of some condition. The exit statement has a syntax which is appropriate for the purpose:

```
exit_statement ::= exit [loop_name] [when condition];
```

However, the raise statement is incomplete in this respect:

```
raise_statement ::= raise [exception_name];
```

CONSEQUENCES:

The incompleteness of the raise statement's syntax hampers the readability of Ada software, and causes some dissonance in the minds of Ada users in that an intuitively "natural" statement (raise Exception when Condition) is not permitted.

WORKAROUNDS: None

POSSIBLE SOLUTIONS:

Revise 11.3.2 to read:

```
raise_statement ::= raise [exception_name] [when condition];
```

EXCEPTION IDENTIFICATION

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

There is no built-in method for determining the name of an exception outside an exception handler. Knowing the last raised exception can be useful in the logic of a sequence-of-statements.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None in the language; workarounds are either compiler-specific or exceedingly cumbersome.

POSSIBLE SOLUTIONS:

Include a function or attribute which returns the name (enumeration literal preferred) of the last exception raised in the callers task or main program. It is important to discriminate based on the "thread of control".

INTERRUPTS

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 11, 13.5.1, 9.6

PROBLEM:

The validation process requires that the vendor "prove" the product. For a vendor to accomplish validation, they need to include in the runtime routines a set of timer utilities and interrupt handlers. The use of the hardware interrupts is a contention item with the applications software for embedded systems (and prohibited for multi-user shared facilities). This causes the user to adapt the interrupts--thereby impacting validation.

The timers and interrupts are used in embedded systems to perform the following requirements: maintain real time for the subsystem, control hardware devices, manage fault monitoring (self-test or builtin tests). The latter makes heavy use of interrupts--with more than one service routine per handler. The compiler vendor can not possibly know the selftest requirements. It is desirable for the user to be able to write interrupt handlers in Ada, but it is undesirable to have the runtime support do so.

The runtime support should not set the realtime clock, exercise machine level interrupts, or enter/exit privileged modes. The notes on interrupts are just not true for all machine models.

IMPORTANCE:

Very high for realtime embedded systems. Almost no importance to host or information systems applications.

CURRENT WORKAROUNDS:

Most vendors have trouble supporting the needs of the application to match the hardware interfaces. Therefore, the programmer must provide more assembler than they care to.

POSSIBLE SOLUTIONS:

1. make sections on timing, machine level exceptions, and interrupts an option. Notes in 13.5.1, subpara. # 5 and 6, are not necessarily correct for all hardware.

2. remove mention of interrupts from LRM and provide wording such that if the hardware can recognize and support the exception, then allow it to do so without any extra special handling in the runtime support. Exception handling does not need to duplicate hardware support functions.

For validation purposes, the compiler could generate exceptions for the host environments or emulators, then as an optimization use application supplied interrupt handlers. Note that the interrupts may be masked in some situations, e.g., fixed point arithmetic.

3. An interrupt handler does not have to have a TASK model, but merely a procedure that the hardware can enter. The task activation records interfere with the structure of the interrupt vector needs on some architectures. Simplify by only requiring that it be a procedure.

SOME PREDEFINED ADA EXCEPTIONS

DATE: October 10, 1989

NAME: PETER T. BRENNAN

ADDRESS: Grumman Data Systems, MS: D12-237
1000 Woodbury Road
Woodbury, NY 11797

TELEPHONE: (516) 682-8431
E-mail: jlithe@ajpo.sei.cmu.edu

ANSI/MILL-STD-1815A REFERENCE: 11

PROBLEM:

Some of the predefined Ada exceptions are too broad, in particular CONSTRAINT_ERROR. As will be shown below, this exception can occur several different ways, leaving the caller at a loss as to what the problem really is, which seriously affects the ability to properly handle such faults.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The language should be modified to provide additional predefined exceptions to disambiguate current predefined exceptions that have a broad range.

JUSTIFICATION/EXAMPLES/WORKAROUNDS:

The following program section, which successfully compiles, is one example that demonstrates the problems with procedure B handling an ambiguous CONSTRAINT_ERROR.

```
subtype RANGE_20 is INTEGER range 0..20;

procedure A(Y,Z : in out RANGE_20) is
        X : RANGE_20 := Y * Z;
begin
        ...
end;

procedure B is
        I,J : INTEGER;
begin
        ...
        A(I,J);
exception
        when CONSTRAINT_ERROR =>
        ...
end;
```

Three possible situations exist:

1) The parameters I and J are invalid. At runtime, the values of I and/or J exceed the range of RANGE_20. A CONSTRAINT_ERROR is raised at the call (procedure A does not begin execution).

2) The parameters I and J are valid, but while executing the code for the declarative region (X := Y * Z), a CONSTRAINT_ERROR is raised and propagated to the caller. (It makes no difference if procedure A had an exception handler or not.)

3) Procedure A successfully enters the begin block, but then raises CONSTRAINT_ERROR on some calculation. If an exception handler is present for CONSTRAINT_ERROR, it is executed. However, CONSTRAINT_ERROR may still be propagated from any of the exception handlers that procedure A may have.

When a call to procedure A raises a CONSTRAINT_ERROR, it may be from one of three possible conditions. Procedure B has difficulties in interpreting CONSTRAINT_ERROR, because the exception has too wide an application. Since procedure B does not know why CONSTRAINT_ERROR was raised, its ability to properly respond to the real cause is extremely limited.

POSSIBLE SOLUTIONS:

The addition of two predefined exceptions, PARAMETER_ERROR and DECLARATION_ERROR would help solve the problem.

For situation #1 above (parameter out of range), the Ada language should be augmented to raise a new predefined exception, PARAMETER_ERROR.

For situation #2 above (CONSTRAINT_ERROR in declarative section), the Ada language should be augmented to raise a new predefined exception, DECLARATION_ERROR.

Both these cases should have very minimal impact on compilers, and are special because the exception handling scope of procedure A is not reached.

NON-SUPPORT IMPACT:

In the interests of developing highly reliable software, handling exceptions in Ada plays an important part. When the ability to properly determine the failure is clouded, the reliability of the recovery algorithms decreases.

Properly responding to an exception hinges on knowing what that exception is first. The above example illustrates a problem with a predefined exception that has at least three unique meanings. Over-generalization of exceptions hinders fault tolerance and overall reliability of the software.

It is important to note that user-defined exceptions can not be used to solve this problem.

EXCEPTION HANDLING

DATE: April 24, 1989

NAME: Bjorn Kallberg
Endorsed by Ada in Sweden, our number AIS031

ADDRESS: Ericsson Radar Electronics
S-164 84 Stockholm
Sweden

TELEPHONE: 46 8 757 35 08
46 8 752 81 72
E-mail: ada_ubk@kiere.ericsson.se

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

In the large information type systems that Ada presently is being used for, the error diagnostic is more important than in pure embedded systems. Run time errors need to be logged, or presented to the user. The name of an exception is not enough, but additional information is needed, such as name of compilation unit where the error occurred, line etc. Perhaps also some user dependent text need to be printed by the application.

Exceptions in Ada define the errors. They should thus be the ideal vehicle for the enhanced error handling. However, exceptions are in practice a very limited type, with almost no available operations. They can thus not be used for this advanced error handling.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Return an error code with every subprogram.

POSSIBLE SOLUTIONS:

Lowest level: Add facilities to get the following
"attributes" of the last raised exception:

'image,
'line_number_where_raised,
'compilation_unit_name_where_raised.

The raise statement without exception name will not change the above attributes. The line_number_where_raised is implementation dependant, ie, it is allowed to return 0, if the program is highly optimized, or it is otherwise unreasonable to return the line number.

EFFECT ON EXISTING PROGRAMS: NONE

EXCEPTIONS SHOULD BE TREATED LIKE OBJECTS OF A TYPE

DATE: October 29, 1989

NAME: S. Tucker Taft

ADDRESS: Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

TELEPHONE: (617) 661-1840
E-mail: stt%inmet@uunet.uu.net
E-mail: taft@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

Currently exceptions cannot be passed as parameters, nor can their image be printed, nor can sets of exceptions be defined, etc. This makes it difficult to handle groups of exceptions, produce meaningful error messages, etc.

IMPORTANCE: IMPORTANT

Without improving the ability to extend, pass, and print exceptions, their usefulness for error detection and reporting will be limited.

CURRENT WORKAROUNDS:

Additional information may be stored into a global just before raising an exception. However, this doesn't work in multi-tasking programs.

POSSIBLE SOLUTIONS:

Exceptions are currently defined in a strange way, distinct from all other concepts of the language. However, it would be very useful to be able to pass an exception to a generic, or a regular subprogram, to tailor the action of the generic/subprogram.

Furthermore, there is no way in the "when others =>" handler to print out the name of the current exception.

Finally, shared generics are difficult to implement partly because exceptions have a somewhat strange semantics, being a sort of compile-time entity using macro-expansion as the basis for generic instantiation.

Many of these problems would be solved by treating the declaration of an exception as the declaration of a new enumeral from a dynamically growing, unordered, enumeration.

The syntax could then be augmented with:

```
excep2 : new excep1;
```

This would declare a new exception excep2 which is a kind of "sub" exception of excep1. Any handler for excep1 would also handle excep2, unless preceded by a handler which explicitly mentions excep2.

A normal exception declaration is equivalent to:

```
excep : new root_exception;
```

and "when others =>" is equivalent to "when root_exception =>".

It would be useful if syntax were provided to designate the underlying type for these exceptions, perhaps <exception_name>'TYPE, plus 'IMAGE and 'VALUE functions, a way to refer to the "current exception" (perhaps as a function in package System called Current_Exception returning Exception'TYPE), and rules established for handlers with exceptions designated by an exception object/parameter rather than an explicit exception (e.g. evaluate exception choices sequentially, choose first handler which matches).

RETRIEVE INFORMATION ABOUT CURRENT EXCEPTION

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

One frequently introduces a "when others=>" exception handler that is only supposed to be executed in the case of an internal error in the program. In this case, one would like the handler to be able to output information about the conditions at the time of the error (e.g. the name of the exception, the location at which it was raised, a stack backtrace, etc.). Implementations may provide implementation-specific facilities for obtaining some of this information, but using these mechanisms detracts from program portability.

It would be useful to have an implementation-independent interface for obtaining whatever exception context information an implementation is able to provide.

A typical "firewall" exception handler might then look something like

```
procedure Foo is
    ...
exception
    when others =>
        Error_Logging.Log
            ("Unexpected exception handled in  F o o :  "  &
            Current_Exception.Info)
end Foo:;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use implementations-dependent mechanisms.

POSSIBLE SOLUTIONS:

IMPROVED EXCEPTION CAPABILITIES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706[11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

Exceptions are a very valuable feature of Ada, but they do not provide quite as much power and flexibility as is desirable. Several things are missing:

* There is no way to get the image of an exception.
* There is no way to move complex exception handling code out of the body of the handler into a procedure.
* Exceptions cannot be used as generic formals.

IMPORTANCE: ESSENTIAL

Some of these missing capabilities are so important that compiler vendors have added non-portable features to address the problem.

CURRENT WORKAROUNDS: NONE (that are portable, anyway).

POSSIBLE SOLUTIONS:

The following changes to the standard, either separately of in aggregate, address the deficiencies listed above:

It should be possible to declare constants and variables of type EXCEPTION. For example:

```
Standard_Exception : exception := Program_Error;
Current_Exception : exception := Constraint_Error;
...
raise Current_Exception;
```

It should be possible to find out which exception has been raised. For example:

```
exception
        when Constraint_Error =>
        ...
        when others =>
                declare
                The_Exception : constant exception :=
                        exception'Current;
        begin
                [do something]
        end;
```

It should be possible to obtain the image of an exception. For example:

```
exception
        when Constraint_Error =>
        ...
        when others =>
                Text_Io.put_Line ("Unexpected exception: " &
                        exception'Image (exception'Current));
```

It should be possible to handle an exception with a procedure. For example:

```
procedure Handle (This_Exception : in exception) is
begin
case This_Exception is
        when Constraint_Error =>
        ...
        when others =>
                Text_Io.put_Line ("Unexpected exception: " &
                        exception'Image (exception'Current));
        end case;
end handle;
  ...
exception
        when others =>
                Handle (This_Exception => exception'current);
```

It should be possible for exceptions to be generic formals (and for them to be supplied as actuals to generic instantiations). For example:

```
generic
        Reaction : exception;
        ...
procedure Some_Generic is...

procedure Do_Something is new Some_Generic (Reaction => Constraint_Error...)
```

COMPATIBILITY:

The proposed solution is quasi-compatible. All previously-compiled code will re-compile successfully unless it contains an implementation-defined'CURRENT attribute for packages (this is not, however, all that likely).

EXCEPTIONS DO NOT CARRY PARAMETERS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
FAX: +46 758 15133

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

It would be very useful if it were possible to supply parameters from the point where an exception is raised to where it is handled.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

State has to be carried around in the unit to signal the exception handler what was going on at the point where the exception took place. This leads to much code being written that (as is the case with most error handling code) tends to obscure the real purpose and function of the unit.

POSSIBLE SOLUTIONS:

Extend the exception declaration (and make it symmetric with respect to procedure declarations)
exception My_Exception (The_Offending_Value : in integer);
NB: no out parameters allowed.

The handler syntax would be:

```
begin
        ...
exception
        when My_Exception (The_Offending_Value : in integer) do
                ...
                end My_Exception;
        when ...
end;
```

My_Exception : **exception;**

would be equivalent to

exception My_Exception;

TIMED EXCEPTIONS

DATE: October 23, 1989

NAME: Ulf Olsson

ADDRESS: Bofors Electronics AB
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 10000
+46 758 15133 (fax)

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

In order to support deadline scheduling, it would be nice to be able to set a timer that generates an exception on the task if the timer runs out before the corresponding reset operation is called (or the timer is set anew).

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Timer tasks (not so efficient); checking what time it is at various places in the code (even worse)

POSSIBLE SOLUTIONS:

```
E: exception;
D: duration;
task T is ... end T;
...
raise E after D; -- in own task
raise E after D in T; --in somebody else
```

IMPROVED EXCEPTION CAPABILITIES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rationale, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95045-3197

TELEPHONE: (408) 496-3706 (11 am--9 pm)

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

Exceptions are a very valuable feature of Ada, but they do not provide quite as much power and flexibility as is desirable. Several things are mission:

There is no way to get the image of an exception.

There is no way to move complex exception handling code out of the body of the handler into a procedure.

Exceptions cannot be used as generic formals.

IMPORTANCE: ESSENTIAL

Some of these missing capabilities are so important that compiler vendors have added non-portable features to address the problem.

CURRENT WORKAROUNDS: NONE

that are portable

POSSIBLE SOLUTIONS:

The following changes to the standard, either separately or in aggregate, address the deficiencies listed above:

It should be possible to declare constants and variables of type EXCEPTION. For example:

```
Standard_Exception  :  exception  := Program_Error;
Current_Exception   :  exception  := Constraint_Error;
```

```
                ...
        raise Current_Exception;
```

It should be possible to find out which exception has been raised. For example:

```
        exception
                when Constraint_Error =>
                        ...
                when others =>
                        declare
                                The_Exception :         constant exception :=
                                        exception'Current;
                        begin
                                [do something]
                        end;
```

It should be possible to obtain the image of an exception. For example:

```
        exception
                        when Constraint_Error =>
                                ...
                        when others =>
                                Text_Io.Put_Line ("Unexpected exception: "&
                                        exception'Image (exception'Current));
```

It should be possible to handle an exception with a procedure. For example:

```
        procedure Handle (This_Exception  :  in exception) is
        begin
                case This_Exception is
                        when Constraint_Error =>
                                ...
                        when others =>
                                Text_Io.Put_Line ("Unexpected exception: " &
                                        exception'Image (exception'Current));
                end case
        end Handle;
                ...
        exception
                when others =>
                                Handle (This_Exception => exception'current);
```

It should be possible to handle all exceptions declared in a particular package with the following construct:

```
exception
        when Some_Package'exceptions =>
                                --Handle all exceptions declared in Some_Package.
                                ...
        when Some_Package.Some_Nested_Package"exceptions =>
                                --Handle all exceptions declared in
                                Some_Nested_Package.
```

...

It should be possible for exceptions to be generic formals (and for them to be supplied as actuals to generic instantiations). For example:

```
generic
        Reaction : exception;
        ...
procedure Some_Generic is ...

procedure Do_Something is new Some_Generic (Reaction => Constraint_Error...)
```

COMPATIBILITY:

The proposed solution is quasi-compatible. All previously-compiled code will re-compile successfully unless it contains an implementation-defined 'Current attribute for exceptions or an implementation-defined' Exceptions attribute for packages (neither possibility is all that likely).

DETERMINING THE NAME OF AN EXCEPTION WITH A HANDLER

DATE: June 20, 1989

NAME: David F. Papay

ADDRESS: GTE Government Systems Corp.
P.O. Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 11

PROBLEM:

Within an exception handler, it is often useful to obtain the string representation of the exception which caused the transfer to the handler. The language should provide a way to obtain this string.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Name only one exception per exception choice. This would allow the following:

```
EXCEPTION_NAME : STRING (1..20) := (others => ' ');
...

exception
    when CONSTRAINT ERROR =>
        EXCEPTION_NAME (1..16) := "CONSTRAINT_ERROR";
            ...

    when STORAGE ERROR =>
        EXCEPTION_NAME (1..13) := "STORAGE_ERROR";

    when TEXT IO.NAME_ERROR=>
        EXCEPTION_NAME (1..18) := "TEXT_IO.NAME_ERROR";
            ...
            -- etc.
```

POSSIBLE SOLUTIONS:

A solution to this is to support the attribute IMAGE for exceptions within exception handlers. This would allow multiple exception to be handled with the same handler. The reserved word exception could be the prefix for the attribute, and it represents the exception that cause the transfer to the innermost enclosing handler.

```
when CONSTRAINT_ERROR | STORAGE_ERROR=>
        EXCEPTION_NAME (1..exception'IMAGE'LENGTH) := exception'IMAGE;
```

This use of the reserved word 'exception' as the prefix for the IMAGE attribute should only be permitted within an exception handler (but not within the sequence of statements of a subprogram, package, task unit, or generic unit, enclosed by the handler. of. raise statement, 11.3(3)).

Note that this attribute could also be used if exception types are added to the language (see revision request #8905240101, which I submitted).

```
exception
        when in PREDEFINED_EXCEPTIONS =>
                EXCEPTION_NAME (1..exception'IMAGE'LENGTH) := exception'IMAGE;
                -- This would yield the name of the specific exception object
                -- not the name of the exception type.
```

Some points to be worked out:

1. Does this attribute return fully expanded names, or just the simple. name of the exception?

2. What happens if the exception name is anonymous (and was handled by a 'when others' handler?

3. What about the attributes WIDTH and VALUE, which are associated with the IMAGE attribute?

REQUIREMENT TO HAVE ADA SYSTEMS PROPAGATE EXCEPTION IDENTIFICATION AND HANDLING INFORMATION IN A DISTRIBUTED/PARALLEL/MULTIPROCESSOR ENVIRONMENT

DATE: August 25, 1989

NAME: V. Ohnjec (Canadian AWG #012)

ADDRESS: 240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 11.1, 11.2, 11.4.1, 11.4.2, 11.5

PROBLEM:

Exception handling within a Distributed/ Parallel/Multi-processor environment is not currently defined in the Ada LRM. Identification of exceptions out of scope must be addressed as well as propagation path.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUND:

Multi-linguistic and vendor specific solutions exist.

POSSIBLE SOLUTIONS:

The Ada LRM should be updated.

DISCRIMINATION OF PREDEFINED EXCEPTIONS

DATE: September 27, 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraes AG,
8052 Zuerich, Switzerland

TELEPHONE: (001 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 11.1, 11.14

PROBLEM:

The granularity of the predefined exception is too coarse. Sometimes it is necessary, to handle some kind of constraint_error (say null-access) or some kind of layout_error (say line-too-long) locally, and propagate other error situations. This cannot currently be implemented using language defined features.

IMPORTANCE: IMPORTANT

Exception processing is an important part of mission- critical real-time software. The Ada language does not offer sufficient support in this area. A programmer is thus tempted to use non-portable, implementation-dependent features.

POSSIBLE SOLUTIONS:

Many different solutions are possible. To keep the impact on existing implementation small, I would suggest to extend the packages system and io_exceptions. Types and functions as e.g.

```
TYPE exception_cause_type IS
    (range_check, null_access, ...);

SUBTYPE constraint_error_causes IS exception_cause_type  RANGE...;
   SUBTYPE program_error_causes IS ...
...

FUNCTION exception_cause RETURN exception_cause_type;
```

have to be provided.

ELIMINATE NUMERIC_ERROR

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 11.1(6)

PROBLEM:

If it is decide (and this seems like a very good idea; see AI-00387) that all operations that formerly raised Numeric_Error (e.g. division by zero) should instead raise Constraint_Error, then the predefined exception Numeric_Error should be deleted form package Standard.

This will make it easier to detect errors introduced by this incompatible change at compile time instead of at runtime.

Alternatively, Numeric_Error might be declared as a rename of Constraint_Error.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS: NONE

ALLOW EXCEPTION HANDLER TO COLLECT ALL EXCEPTIONS INTO ONE "WHEN" STATEMENT

DATE: June 16, 1989

NAME: Ivar Walseth

ADDRESS: Kjell G. Knutsen A/S
Box 113
4520 Sor-Audnedal
NORWAY

TELEPHONE: +47 43 56205

ANSI/MIL-STD-1815A REFERENCE: 11.2, paragraph 2.

PROBLEM:

Some packages have a lot of defined exceptions which describes the actual problem into details. Suppose the user of this package does not care about anything but the fact that something exceptional happened in the package. It would then be nice and readable with an exception handler collecting all these exceptions into one "when" statement.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Handle each exception individually.

POSSIBLE SOLUTIONS:

Extend the definition of section 11.2, paragraph 2 to

```
exception_choice ::    =exception_name.others|
                           package_simple_name.others|
                           others
```

This would allow the following:

```
exception
when matrix.others | complex.others => RAISE computing_error;
end;
```

EXCEPTION_NAME

DATE: October 20, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 11.2(5)

PROBLEM:

In an Ada program, it is not possible to print a message in an EXCEPTION WHEN OTHERS clause which indicates the name of the exception, nor is it possible to determine where the exception was raised if not caught in each procedure. Some compilers provide this information in an operating systems traceback if no exception handler is specified, but the output is generally not very neat.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Make every procedure in the program have an exception handler which provides some kind of traceback. This, however, causes a lot of extra unnecessary code, especially when some of these exceptions "can't" happen.

POSSIBLE SOLUTIONS:

Provide a way for the programmer to access the exception stack, and require all compilers to put exception information on this stack, provided that the programmer requests it.

1. Introduce pragma Exception_trace_stack(on); The alternate option is "off". If on, then whenever a procedure raises an exception, the exception handler of that procedure will have an implicit WHEN OTHERS clause if one is not explicitly stated. This will push certain exception ii formation on the stack, as described below. Also, if a WHEN OTHERS clause does exist and contains a raise statement, the same thing will happen.

2. Introduce a package EXCEPTION_TRACING, which has procedures which can access:

   (a) the fully qualified name of the exception,
   (b) the fully qualified name of the procedure which raised the exception,
   (c) the number of the line in the program source which raised the exception.

   The EXCEPTION_TRACING client will normally print out this information in some readable

format.

This solution is most valuable during the testing phases of a program, where programs usually work, but not all the time. Not having an exception handler is dangerous, but having an EXCEPTION WHEN OTHERS in the main program is even more so, since then no information is available.

PRIVATE EXCEPTIONS

DATE: August 7, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd.
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 11.3

PROBLEM:

The writer of a package of related operations and structures will often wish to indicate misuse of the package through the raising of exceptions declared in the package. Such exceptions will be made visible to the user so that the user can then handle them in order to carry out appropriate recovery actions.

Unfortunately there is nothing to prevent the user from directly raising such an exception (perhaps by mistake) in an irrelevant situation. This means that the provider of the package cannot guarantee to the user that the exceptions will only occur in stated circumstances.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

An exception declared in a package specification could be tagged in some way (e.g., marked private). This would mean that the exception could not be used outside the package explicitly in a raise statement. It would still be allowed to be reraised in a handler by a raise statement without an exception name.

REFERENCE:

See Exercise 10.2(4) in Programming in Ada by Barnes (3rd edition).

NAME OF THE "CURRENT EXCEPTION"

DATE: October 13, 1989

NAME: David A. Smith, on behalf of SIGAda Ada Language Issues Working Group

ADDRESS: Hughes Aircraft Company, A1715
16800 E. Centre Tech Parkway
Aurora, CO 80011

TELEPHONE: (303) 344-6175
E-mail: dasmith @ ajpo.sei.cmu.edu or
E-mail: smith @ cel860.hac.com

ANSI/MIL-STD-1815A REFERENCE: 11.4

PROBLEM:

The following comment represents discussion that took place at the July 1986 meeting of the Ada Language Issues Working Group (ALIWG). It has been noted by many users that it is frequently desirable in handling an exception to have access to a string representing the exception's name. This is useful for debugging purposes and for building diagnostics into a program. While this feature is available in some implementations, the consensus was that it should be required of all implementations (like, for example, T'IMAGE for enumeration types).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Workarounds for this capability are awkward or impossible to build.

POSSIBLE SOLUTIONS:

Many variations on this capability were discussed.

1) The string value could be accessed in a variety of ways:

a)
```
Put( Exception'Image);                                    -- attribute
-- This odd use of a reserved word seems unnecessary. There doesn't seem to be anything
-- appropriate for this string to be an attribute of -- it would be meaningful as an
-- attribute of a task, but the main task in particular is not nameable.
```

b)
```
Put( Standard.Exception_Name);                            -- function call
Put( System.Exception_Name);                              -- function call
Put( Current_Exception.Exception_Name);                   -- function call
Put( Exception_Name);                                     -- function call
-- It was agreed this string should be returned by a zero-argument function; the function
-- could be its own library unit could be exported by an ad hoc package.
-- Packages Standard and System both seem inappropriate.
```

-- Agreement on subprogram and/or package names was lacking.

2) The string returned could be one of a number of things:

a) The simple name of the exception
b) The "fully qualified" name (this is not well defined)
c) The simple name plus the line number and compilation unit where the exception is defined

The sentiment favored more information than the simple name of the exception, but neither option b nor c is very satisfactory. A possible extension of choice b would be to conventionalize a name-like substring for anonymous blocks:

"PACKAGE_NAME.PACKAGE_NAME.<ANONYMOUS_BLOCK>.EXCEPTION_NAME"

3) Two proposals were considered for the scope for invoking this function:

a) The function returns a non-null string when called from a point where the "RAISE;" statement is allowed (ie, without an explicit exception name). If called elsewhere, returns a null string.

b) The function can be called anywhere and returns a null string or "the most recent exception whose handling has not been completed", (assuming this is well defined).

It was suggested that syntactically restricting the point of call (option a) might be necessary in some implementations.

Option b was felt to be more "user friendly". *A variation would be to return "the most recently raised* exception, if any".

TASKS DIE SILENTLY

DATE: March 3, 1989

NAME: Bjorn Kallberg
Endorsed by Ada in Sweden. Our number AIS-032

ADDRESS: Ericsson Radar Electronics
S-164 84 Stockholm
Sweden

TELEPHONE: 46 8 757 35 08
46 8 752 81 72
E-mail: ada_ubk@kiere.ericsson.se

ANSI/MIL-STD-1815A REFERENCE: 11.4.1 (9)

PROBLEM:

A non handled exception in a task causes the task to die silently (excluding the special case of a rendezvous). In all other parts of the language, a great concern has been made towards secure program. Thus, in the other parts no errors will be unnoticed except when so stated using exception handlers. The silent completion of tasks is against the security principle otherwise adhered to.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

A non handled exception in a task shall raise an asynchronous exception in the parent task.

EFFECT ON EXISTING PROGRAMS:

Very small. If any program is relying on tasks to die, (which I doubt, and which certainly is not good style) then an exception handler has to be added to the task body (when others=>null)

REMOVE CANONICAL ORDERING RULES FOR THE ASSIGNMENT OPERATOR

DATE: September 30, 1989

NAME: Ron Lieberman

ADDRESS: CONVEX Computer Corporation
3000 Waterview Pkwy
P.O. Box 833851
Richardson, TX 75083-3851

TELEPHONE: (214) 497-4248
E-mail: lieb@convex.com

ANSI/MIL-STD--1815A REFERENCE: 11.6

PROBLEM:

The current wording of 11.6(3) does not allow implementation to legally perform certain optimization such as vectorization of loops, code motion, and optimal instruction scheduling.

IMPORTANCE: ESSENTIAL

Ada compilers are forced to either not perform useful optimizations, or elect to perform these optimizations only when directed to do so by the presence of a PRAGMA or command line request for optimization. Note that these optimizations are generally not performed when undergoing validation.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

1) Remove the canonical ordering rule for operators.

2) Relax the canonical ordering rule for operators when optimization is requested for a compilation.

ALLOWED OPERATION REPLACEMENT

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.,
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 11.6

PROBLEM:

Allowed replacements of operations need to be clarified.

It is unclear from 11.6, whether replacement of operation sequences by mathematically equivalent, but computationally different operations is allowed. As a simple example, can "-k+A" (k constant, A a floating point object) be computed as "A-k" ? On some architectures, one of these alternatives may overflow while the other one does not.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

CANONICAL EXECUTION ORDER

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.,
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE): 11.6

PROBLEM:

As architectures with micro-parallelism at the instruction level become more prevalent (e.g., instruction pipelining, instructions performing parallel operations, vectorizing, parallel flt.point units, etc.), a narrow notion of execution order in a programming language becomes a major hindrance to the generation of optimal code.

Chapter 11.6 provides too little freedom for parallelizing and reordering the instruction stream in the potential presence of exceptions. At the same time, the rules as stated are largely incomprehensible to the average user of Ada, leading to misconceptions about the validity of code optimizations and consequently to mistrust of optimizing compilers. Finally, they are sufficiently vague to confuse implementors of Ada.

Interpreted by the letter, 11.6(4) could be understood as preventing any parallelization of instructions except within a single expression.

Such interpretation would considerably weaken, if not destroy, the competitive position of Ada vis-a-vis other languages for many target architectures.

The following examples illustrate some of the problems. They are expressed in terms of problems with overflow exceptions; other constraint error situations can be equally substituted.

For any architecture with instruction parallelism or pipelining:

```
        B := 3;
        A := 2**16 * A;        -- (1) overflows and raises CONSTRAINT_ERROR
        B := 5;                -- (2)
    ....
exception
        when CONSTRAINT_ERROR =>
                if B = 5 then Failed; end if; --- ???
```

Synchronizing the pipeline for every instruction that might overflow is a major code quality penalty. Not being able to perform (1) and (2) in parallel, if the architecture supports it, is a major penalty.

Another example is:

```
        A := F * K;    -- (1) always ok, but cycle-intensive
        B := B/Zero;   -- (2) division by zero
    ....
exception
    when others =>
            if A /= F * K then Failed; end if; --- ???
```

Can (2) be executed before (1) completes ? If not, major speed penalties may arise. Note that, on some architectures, floating point operations may take a large number of cycles, during which a considerable amount of subsequent instructions could be executed, before the fl.pt. operation completes or fails.

For vectorizing architectures:

```
for i := 1 to 40 loop
        a(i) := b(i) + c(i);       -- overflows for i = 35
end loop;
exception
        when constraint_error =>
                if A(34) /= B(34) + C(34) then Failed; end if;  ???
```

Since it is very difficult to detect the absence of overflow potential at compile time, the example might lead to the conclusion that Ada is not suitable for vectorized architectures at all.

11.6(7) is unclear both in the technical terms used and in its scope. Is assignment a predefined operation ? It is not a predefined operator. Nowhere is it stated that basic operations are predefined operations. The referenced sections 3.3 and 3.3.3 do not provide a definition of the term. Peripherally, what is the definition of "rendering ineffective" in 11.6(7) ? The entire area of removing useless, but potentially exception raising code needs to be clarified.

Note that, if assignment can partake in reordering, then the notion of retaining a semblance of a canonical execution order prescribed by the language becomes seriously inconsistent.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Allowed variances in the execution order of operations should be oriented on data flow dependencies and not on an enumeration of special rules for deviating from the canonical execution order.

A better paradigm for explaining allowed reorderings and omissions might be to start from the notion of data dependency (rather than allowed reordering in the canonical execution order).

The fundamental rules would then be that

1. any operation directly affecting the input values of a subsequent operation in the canonical execution order must be executed and completed prior to the latter one.

2. Any operation influencing neither control flow nor input values of other performed operations (other than by implicit timing effects) need not be performed.

3. Any operation whose only effect is the implicit raising of a predefined exception need not be performed.

The notion of operation should definitively extend to assignment and parameter passing and possibly even subprogram calls (with suitable refinements of the rules above).

It then remains to provide rules that restrict earliest and latest point of execution of operations. Such rules presumably would at least guarantee that operations raising exceptions are executed within the same frame, accept statement and body as in the canonical execution order. It is arguable whether operations guaranteed to not raise exceptions should be allowed to be executed earlier or later than this rule would otherwise allow. Invariant hoisting and peephole optimizations for pipelined architectures could be substantially affected by this latter ruling.

ALLOW VECTORIZATION

DATE: October 30, 1989

NAME: J. M. Holdman, A. R. Leifeste

ADDRESS: Shell Development Company
P.O. Box 481
Houston TX, 77001

TELEPHONE: (713) 663-2260
E-mail: jon@shell.com

ANSI/MIL-STD-1815A REFERENCE: 11.6

PROBLEM:

In producing efficient code for vector machines, today's FORTRAN compilers often perform rather dramatic code reordering. This includes such things as loop unrolling, loop interchange, loop fission, loop fusion, and many others. Section 11.6 allows reordering involving predefined operators and basic operations (except assignment). However, the legality of the type of code movement done during vectorization is unclear.

IMPORTANCE: ESSENTIAL

The usability of Ada on vector machines, such as Cray, Convex, IBM 3090 and DEC Vector Facilities, will be seriously limited if efficient vectorization (which likely requires this type of code reordering) cannot be done.

CURRENT WORKAROUNDS:

Using pragma interface to FORTRAN.

POSSIBLE SOLUTIONS:

Add an additional allowable value for pragma optimize, such as VECTORIZE.

RELAX CANONICAL ORDERING RULES TO ALLOW REORDERING ASSIGNMENT STATEMENTS

DATE: October 30, 1989

NAME: J. M. Holdman, A. R. Leiserste

ADDRESS: Shell Development Company
P.O. Box 481
Houston TX, 77001

TELEPHONE: (713) 663-2260
E-mail: jon@shell.com

ANSI/MIL-STD-1815A REFERENCE: 11.6 (3-4)

PROBLEM:

In many applications, such as scientific computations, good performance is essential. Where speed of execution is of primary importance, the results of an operation where an exception is raised are usually discarded. Knowing that an exception was raised as some point during the operation is usually sufficient. While many optimization and vectorization techniques traditionally used for these applications involve reordering assignments, the Ada Reference Manual explicitly excludes assignment statements when discussing reordering of basic operations and predefined operators.

IMPORTANCE: ESSENTIAL

Ada compilers are unable to perform useful optimizations and vectorization, even if PRAGMA OPTIMIZE (TIME) is specified. This results in Ada software being very inefficient for certain application domains, such as compute intensive numerical and scientific processing, and inhibits the utilization of Ada in these areas.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

1. Remove the phrase "other than assignments" from the first sentence of 11.6(3).

2. Add an additional operand to PRAGMA OPTIMIZE to allow the programmer to request that this rule be relaxed.

FORCING LOCAL EXCEPTION--CHECKS

DATE: September 17, 1989

NAME: Ivar Walseth

ADDRESS: Kjell G. Knutsen A/S
P.O. Box 113
N-4520 Sor-Audnedal
NORWAY

TELEPHONE: +43-56205

ANSI/MIL-STD-1815A REFERENCE: 11.7

PROBLEM:

To increase execution speed and to decrease program size, one is sometimes forced to suppress runtime checks. To achieve this, the pragma SUPPRESS is used.

For many program units, this suppressing is OK. The problem arises when parts of the code depend on exceptions raised from runtime checks in the implementation of the program. These modules may then be tested and found OK. Later in the lifetime of the module, somebody may include pragma SUPPRESS in the module.

For example, there may be a multiplication that should not overflow except in rare "catastrophic" cases that are best handled as exceptions (i.e., the program is using exceptions as exceptions, rather than using exceptions as debugging aids). The handler is buried deep in a subprogram in a large package: and overlooking the dependence on the check, pragma SUPPRESS is introduced for the whole body by placing it just after the procedure body heading.

A particularly insidious form of the problem arises with the (many) compilers that have a command-line switch to suppress checks: after a recompilation using that switch there's not even a textual record in the source of the suppression: and this form of suppression is particularly easy to do thoughtlessly, without a careful inspection of the code.

IMPORTANCE: IMPORTANT

for the high-reliability community

CURRENT WORKAROUNDS:

1. Avoid suppressing at all; or
2. be very careful when suppressing (especially in the command-line switch case); or
3. avoid constructions relying on runtime checks. Workaround #3 is frequently unrealistic: it is (for example) very difficult to determine before a multiplication that it will cause overflow, yet the multiplication is needed for the application

Some compilers warn about an exception handler when all the checks corresponding to it are suppressed, but:

that doesn't cover the case where not all checks are suppressed;

that doesn't cover the case where the exception is to be propagated;

warnings are easily overlooked.

POSSIBLE SOLUTIONS:

Include a new pragma to ensure that a check is performed in a block, even if it is suppressed in a surrounding block:

```
package body MOSTLY_NOT_DEPENDENT_ON_EXCEPTION_CHECKS is pragma SUPPRESS
(RANGE_CHECK);
        -- Introduced late in the development process

    -- many procedures

    procedure HIGHLY_DEPENDENT_ON_EXCEPTION_CHECKS is
        pragma DON'T_SUPPRESS(RANGE_CHECK); -- or pragma CHECK
            -- Introduced early in the development process,
            -- partly out of prudence, partly to document
            -- the intended dependence

    begin
        ...     -- code that does a multiplication, or whatever
    exception
        when CONSTANT_ERROR => ...

    end HIGHLY_DEPENDENT_ON_EXCEPTION_CHECKS;

    -- many procedures

end MOSTLY_NOT_DEPENDENT_ON_EXCEPTION_CHECKS;
```

For additional references to Section 11. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

NUMBER	TITLE	PAGE
0036	MAKE EXCEPTION A PREDEFINED TYPE	3-16
0111	FAULT TOLERANCE	5-2
0200	WHEN WITH RETURN AND RAISE	5-53
0240	UNRESTRICTED COMPONENT ASSOCIATIONS	4-52
0254	IMPLICIT RAISING OF EXCEPTIONS FOR INTERMEDIATE COMPUTATIONS	3-159
0362	RAISEWHEN	5-44
0432	IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (II)	3-124
0651	EXCEPTIONS RAISED ON OTHER TASKS	9-76
0665	DISTRIBUTED/PARALLEL SYSTEMS	9-96

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 12. GENERIC UNITS

POSSIBLE SOLUTIONS:

As suggested above, a proposed syntax would be:

```
type A(<>) is private;      -- Unconstrained allowed
type A() is private;        -- Unconstrained disallowed (unless has
                            -- discriminant defaults)
```

Another possible syntax is to allow a representation-like clause -- "for A'constrained use False" or "for A'constrained use True" for allowed/disallowed, though it is a little misleading since the real rule does not disallow unconstrained types so long as there are defaults for discriminants, and unconstrained is allowed, it is not actually required (presumably).

In any case, if unconstrained is allowed, then at compile time of the generic spec and body, use of the formal type would be illegal in circumstances requiring a constrained type (or defaults for the discriminants).

If unconstrained is disallowed, then at compile time of the instantiation, it would be illegal to specify an unconstrained subtype (unless the discriminants had defaults).

This solution (or equivalent) would remove undesirable dependencies between generic bodies and their instantiations, and would simplify legality checking in the compiler at both instantiation time and body recompilation time, since there would be no need to search the library for other units which might be made illegal by the current unit.

Upward compatibility concerns mean that the "non-committal" syntax "type A is private" should remain supported, though probably "denigrated" and intended for removal in Ada 200x.

Another benefit of allowing generic code sharing is increased flexibility in compilation orders. A code sharing generic implementation generally can allow recompilation of generic bodies at any time without penalty - requiring instantiations to be recompiled is not necessary.

CURRENT WORKAROUNDS:

If code sharing is not implemented when it is needed, often the only solution is to avoid using generic units.

POSSIBLE SOLUTIONS:

Do not implement proposals that cannot reasonably be implemented in a code-shared generic unit. In particular, do not implement new features in Ada that require a generic body to know information given in an instantiation which must be static. (For example, do not allow generic formal types and their attributes in static expressions inside the generic body).

Ideally, one should insure that the contract model is adhered to in Ada 9X, eliminating cases where the legality of an instantiation depends on the body provided. Insure that rules allowing generic bodies to be compiled without affecting the instantiation are maintained.

SUBMITTED FOR DESCRIBING OBJECTS DISTRIBUTED ACROSS GENERICITY OF GENERICS

DATE: July 20, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, UK

TELEPHONE: +44-491-579090

ANSI-MIL-STD-1815A REFERENCE: 12

PROBLEM:

The overall philosophy of generic units is to allow the writing of a unit which through parameterization and subsequent instantiation can generate a particular actual unit. Thus the instantiated unit should have the types and names chosen by the user.

Currently, however, any name exported by a generic package (by being declared in its specification) is established when the package is written and cannot be changed by parameterization.

Consider typically

```
generic
        type BASE is (<>);
package SET_OF is
        type SET is private;
        type LIST is array (POSITIVE range <>) of BASE;
    EMPTY, FULL: constant SET;
    ...

end SET_OF;
```

The names SET, LIST, EMPTY AND FULL are permanently fixed an cannot be chosen at instantiation. It is therefore not possible to choose appropriate names for these exported entities according to the abstraction represented by the actual type corresponding to the formal type BASE.

IMPORTANCE: IMPORTANT

The workarounds described below are not entirely satisfactory since they leave old names cluttering the name space and thus cause confusion with attendant risk of error. A proper solution is required if the full benefit of the abstraction properties of the generic mechanism is to be obtained.

CURRENT WORKAROUNDS:

Different names can be given to exported names by renaming or through the use of the derived type mechanism. In both cases, however, the old names are still visible and thus the proper abstraction is not achieved.

POSSIBLE SOLUTIONS:

A further parameter mechanism for generics enabling the parameterization of simple names would suffice.

REFERENCE:

See Programming in Ada by Barnes (3rd edition) pp 258 et seq.

IMPROVED SUPPORT FOR PROGRAM COMPOSITION AND OBJECT-ORIENTED PROGRAMMING TECHNIQUES

DATE: September 26, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 12

PROBLEM:

The re-export mechanisms in Ada are too limited to support program composition well.

In particular:

1. There is no way to import named numbers, record types, generic units, and exceptions into a generic unit, so there is no way to re-export them either;

2. There is no way to retain the relationship between subtypes of the same type or between types derived from the same type in generic formal parameters;

3. There is no efficient way to re-export a type and all of its attributes and operations;

4. Generic formal types and objects which are re-exported from an instance of a generic unit lose their staticness; and

5. There is no way to import related entities as a group (e.g., package) and "pass-through" such groups of entities to the instantiator of a generic package.

These deficiencies are impediments to the reusability of Ada library units, to the composition of larger abstractions from library units, and to the object-oriented programming style, whether or not Ada 9x provides more direct support for object-oriented programming. See, for instance, "Composable Ada Software Components and the Re-Export Paradigm", Ada LETTERS,
VIII.1-58 and "Using the Re-Export Paradigm to Build Composable Ada Software Components", Ada LETTERS, VIII.2-39 for a detailed discussion of the ways in which Ada is less capable than it should be for program composition.

IMPORTANCE: IMPORTANT for most applications

ESSENTIAL for support of object-oriented programming techniques. These facilities should be provided even if no more comprehensive approach toward the support of object-oriented programming is incorporated into Ada 9X.

CURRENT WORKAROUNDS:

For importing named numbers, it is possible to use a generic formal type and a corresponding generic formal object. However, requiring a specific numeric type results, in general, in loss of precision, and requires inventing a type just for that purpose. It is impossible to re-export a named number from a generic unit.

For importing (and re-exporting) record types, generic units, and exceptions there are no workarounds.

For the relationship between subtypes or derived types, an approach that sometimes is feasible is to define the subtypes or derived types in the generic unit, rather than importing them, but this technique requires drastic restructuring of the design of the software, is not transitive (i.e., can only be applied at one level of generic instantiation), and is not universally applicable.

For re-exporting a type in its entirety, a subtype declaration plus explicit renaming declarations for all of the desired operations may be used. However, this is so tedious as to effectively deny the re-export of a numeric type to all but a determined few programmers. Please see the papers referred to above for a detailed discussion, particularly about why derived types do not solve the problem referred to here.

For the loss of staticness, there is no workaround.

For the inability to import and re-export groups of entities, there is no workaround.

POSSIBLE SOLUTIONS:

1. Extend generic formal parts by adding:

   generic formal named numbers,
   generic formal exceptions,
   generic formal subtypes,
   generic formal derived types,
   generic formal record types,
   generic formal packages,
   generic formal generic subprograms, and
   generic formal generic packages.

Generic formal subtypes and generic formal derived types would be related to their base type and parent type, respectively through identifiers in the following way for discrete types and integer types:

```
type T is (<>);

subtype S is T (<>);          --  Can only match a subtype of the discrete type which matches the
                                  generic formal discrete type with identifier T

type D is new T (<>);         --  Can only match a type derived from the discrete type which matches
                                  the generic formal discrete type with identifier T

type T is range <>;
```

subtype S **is** T **range** <>; -- Can only match a subtype of the integer type which matches the generic formal integer type with identifier T

type D **is new** T **range** <>; -- Can only match a type derived from the integer type which matches the generic formal integer type with identifier T

Similarly, for fixed and floating point types:

type T **is delta** <>;

subtype S **is** T **delta** <>; -- Can only match a subtype of the fixed point type which matches the generic formal fixed point type with identifier T

type D **is new** T **delta** <>; -- Can only match a type derived from the fixed point type which matches the generic formal fixed point type with identifier T

type T **is digits** <>;

subtype S **is** T **digits** <>; -- Can only match a subtype of the floating point type which matches the generic formal floating point type with identifier T

type D **is new** T **digits** <>; -- Can only match a type derived from the floating point type which matches formal floating type with identifier T

For private types, we might have:

type T **is** [limited] **private;**

subtype S **is private** T <>; -- Can only match a subtype of the private type which matches the generic formal private type with identifier T

type D **is new private** T <>; -- Can only match a type derived from the private type which matches the generic formal private type with identifier T

2. Provide a means for importing record types (which would then be re-exportable). For example assuming T, S, and D are generic formal (sub)types:

```
generic
        type T is range <>;

        type R is
                record
                        C1 : T;
                end record;
package P is ... end P;
```

3. Provide a means for passing generic units through a generic unit. Within the generic unit the only allowed use of the identifier of the generic formal generic parameter would be to re-export it. That is, the generic formal parameter can't be instantiated in the generic unit. For example, we might have:

```
with Unchecked_Conversion;
generic
        with generic function F_In is <>;
        with generic function G_In is Unchecked_Conversion;
        with generic procedure P_In (I : in Integer);
        with generic package Q_In;
package Re_Export is
        generic
        function F renames F_In;

        generic
        function G renames G_In;

        generic
        procedure P (I : in Integer) renames P_In;

        generic package Q renames Q_In;
end Re_Export;
```

4. Provide a single declaration which can re-export a type and all of its operations. For instance,

```
package P is
        type T is ...;
end P;

with P;
package Q is
type U renames P.T;     -- This has a similar effect as derivation would have, in that all of the
                        attributes and operations of P.T are available for type U, but the *type*
                        of  U is the same as the type of P.T.
end Q;

with Q; use Q;
procedure R is
... -- R has full visibility of T and all of its operations.
```

5. Preserve the static properties of generic actuals which are re-exported from a generic package using the generic formals so that if the generic actuals are static, they are considered static in the instantiating context also (although of course not static inside the generic itself).

A more comprehensive example of the usage of these features in concert might be as follows:

```
generic

        Pi_In : constant := 3.14;          -- With a default value

        Error_In : exception is <>;
        End_Error_In : exception is Text_IO.End_Error;
                -- With a default value

        type Int_In is range <>;
```

```
        subtype Sub_Int_In is Int_In range <>;
                -- The actual may be any subtype of
    Int_In

        type Derived_Int_In is new Int_In range <>;
                -- The actual may be any type derived from Int_In

        type Enum_In is (<>);
        subtype Sub_Enum_In is Enum_In range (<>);
                -- The actual may be any subtype of Enum_In

        type Derived_Enum_In is new Enum_In range (<>);
                -- The actual may be any type derived from Enum_In

        type Fixed_In is delta <>;
        subtype Sub_Fixed_In is Fixed_In delta <>;
                -- The actual may be any subtype of Fixed_In

        type Derived_Fixed_In is new Fixed_In delta <>;
                -- The actual may be any type derived from Fixed_In

        type Floating_In is digits <>;
        subtype Sub_Floating_In is Floating_In digits <>;
                -- The actual may be any subtype of Floating_In

        type Derived_Floating_In is new Floating_In delta <>;
                -- The actual may be any type derived from Floating_In
```

Any actual package which provides matches for each of the generic formals declared here will match this generic formal package parameter:

```
    with package P_In
            type T is range <>;
                    -- Any generic parameter declaration may be used here ... Additional generic
                    parameter declarations
    end P_In;

package G is

    Pi : constant renames Pi_In;

    Error : exception renames Error_In;
    End_Error : exception renames End_Error_In;

    type Int renames Int_In;            -- Re-exports Int_In and all of its operations.
    subtype Sub_Int renames Sub_Int_In;
    type Derived_Int renames Derived_Int_In;

    type Enum renames Enum_In;
    subtype Sub_Enum renames Sub_Enum_In;
    type Derived_Enum renames Derived_Enum_In;
```

```
        type Fixed renames Fixed_In;
        subtype Sub_Fixed renames Sub_Fixed_In;
        type Derived_Fixed renames Derived_Fixed_In;

        type Floating renames Floating_In;
        subtype Sub_Floating renames Sub_Floating_In;
        type Derived_Floating renames Derived_Floating_In;

        package P renames P_In;

end G;


package Q is

        type Derived_Integer is new Integer;

        type Color is (White, Gray, Black, Red, Green, Blue);
        subtype Gray_Scale is Color range White .. Black);
        type Derived_Color is new Color range White .. Green;

        subtype Natural_Duration is Duration range 0.0 .. Duration'Last;
        type Derived_Duration is new Duration range 0.0 .. 100.0;

        subtype Natural_Float is Float range 0.0 .. Float'Last;
        type Derived_Float is new Float range 0.0 .. 100.0;

end Q;


package R is

        type Int1 is range 0 .. 10;
        type Int2 is range -1 .. 1;

        Error : exception;

end R;


with G;
with Q;
with R;
with Text_IO;
package New_G is new G (

        Pi_In => 3.141592654,

        Error_In => R.Error,
        End_Error_In => Text_IO.End_Error,
```

```
        Int_In => Integer,
        Sub_Int_In => Positive,              -- Sub_Int_In is static to users of New_G, so, for instance,
                                             it may be  used in static expressions.
        Derived_Int_In => Derived_Integer,

        Enum_In => Q.Color,
        Sub_Enum_In => Q.Gray_Scale,         -- If Gray_Scale is a static subtype, it will also be static for
                                             the users of New_G, so, for instance, it can be used in case
                                             statements.
        Derived_Enum_In => Derived_Color,

        Fixed_In => Duration,
        Sub_Fixed_In => Q.Natural_Duration,
        Derived_Fixed_In => Derived_Duration,

        Floating_In => Float,
        Sub_Floating_In => Q.Natural_Float,
        Derived_Floating_In => Derived_Floating,

        P_In => R(T => R.Int2);      -- Or some such syntax

end New_G;
```

ALLOW PACKAGE NAMES AND EXCEPTIONS AS GENERIC PARAMETERS

DATE: October 29, 1989

NAME: Michael F. Brenner

DISCLAIMER:

The opinions are mine; not necessarily InterACT's

ADDRESS: InterACT Corporation
417 Fifth Avenue
New York, New York 10016

TELEPHONE: (212) 696-3680

ANSI/MIL-STD-1815A REFERENCE: 12

PROBLEM:

To allow maximum reuse of software, package names and exceptions should be passable as generic parameters. I thought this was submitted a long time ago, but I can not find it in the Ada 9X Project Report, so I am submitting it.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Code is not reused as a generic if it has a "with" in it.

POSSIBLE SOLUTIONS:

Allow package names and exceptions to be passed as generic parameters.

GENERIC FORMAL NAMED NUMBERS

DATE: July 21, 1989

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of Ada UK.

ADDRESS: IPSYS plc
Marlborough Court
Pickford Street
Macclesfield
Cheshire SK11 6JD
United Kingdom

TELEPHONE: + 44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 12.1

PROBLEM:

It is not possible to parameterize a generic unit with a static numeric quantity, e.g. a table dimension, time value, hardware address, etc. This severely restricts the classes of reusable components that may be expressed using Ada generic units; for instance, the development of sensible generic fixed point packages is prevented.

IMPORTANCE: IMPORTANT

If Ada is to achieve its true potential as a language for writing reusable components.

CURRENT WORKAROUNDS:

There are no completely satisfactory workarounds, although in some restricted cases the problem may be avoided. For example, consider a generic package being parameterized by the dimension of a table as an integer value:

```
generic
   DIMENSION : in INTEGER;
package TABLE_MANAGER is
   --etc.
end TABLE_MANAGER;
```

Within the package it is not possible to define a (static) range involving DIMENSION. It can however be used in a (dynamic) subtype indication thus:

```
type TABLE_RANGE is new INTEGER range 1 .. DIMENSION;
```

A table dimension outside the range of INTEGER would cause a CONSTRAINT_ERROR on instantiation. Additional packages would need to be defined to make use of any available longer (or shorter) integer types.

An alternative, less convenient, workaround is to define the component so that the dimension range is

parameterized as a discrete type, e.g.:

```
generic
   type DIMENSION_RANGE is (<>);
   DIMENSION : in DIMENSION_RANGE
package TABLE_MANAGER is
   -etc.
end TABLE_MANAGER;
```

Here, the user of the component must provide both a discrete type and the dimension of the table as actual parameters.

A particularly common requirement, where the above workarounds do not work, is for a generic unit parameterized by some hardware address, e.g. a class of device handlers. It is not possible to specify an address clause for a task entry based on a generic formal value.

The only general workaround to this problem is to employ tools which provide some form of macro expansion capability, i.e. the Ada source is written as a template and macro expanded with actual values as required. The disadvantages of this solution are that the potential for code sharing is lost, and that components become dependent on auxiliary tools.

POSSIBLE SOLUTIONS:

The proposed solution is to extend the Ada language to permit generic formal named numbers. The proposed addition to the syntax is:

```
generic_parameter_declaration ::= number_declaration
```

Note that the mandatory static expression for the initialization of the number declaration serves to disambiguate between universal integers and universal reals within the generic unit.

The matching rules for generic actual parameters corresponding to generic formal named numbers are the same as those for number declarations, i.e. the actual parameter must be a static universal expression resulting in the correct type.

As an example:

```
generic
   DIMENSION : constant := 256;
   DEL : constant := 2.0 ** (-15);
   CONTACT_ADDRESS : constant :=16#0020#;
package IMPORT_NUMBERS is
   type TABLE_RANGE is range 1 .. DIMENSION;
   type FRACTION is delta DEL range -1.0 .. 1.0-DEL;

   task HANDLER is
      entry CONTACT;
      for CONTACT use at CONTACT_ADDRESS;
   end HANDLER;
end IMPORT_NUMBERS;
```

GENERIC FORMAL EXCEPTIONS

DATE: July 21, 1989

NAME: Anthony Elliott, from material discussed with the Ada Europe Reuse Working Group and members of Ada UK.

ADDRESS: IPSYS plc
Marlborough Court
Pickford Street
Macclesfield
Cheshire SK11 6JD
United Kingdom

TELEPHONE: +44 (625) 616722

ANSI/MIL-STD-1815A REFERENCE: 12.1

PROBLEM:

It is not possible to parameterize a generic unit with an exception declaration. This makes it very difficult for the implementation of the generic unit to handle exceptions that may be raised by subprogram parameters.

The use of Ada for object-oriented design or for expressing Abstract Data Types (ADTs) results in packages that are characterized by providing one or more (usually private) types, subprograms to operate on those types, and exceptions which may be raised by the subprograms. It is often desirable to parameterize one package (or abstraction) with another.

Although Ada allows for dependencies between packages to be expressed through the context clause, this results in a named dependence on a specific package. By making use of generic units, dependencies on other packages can be expressed through generic formal parameters. This allows the package to be compiled independently of packages on which it depends, and for different configurations of the package to be built (i.e. instantiated) within the same Ada program. Full parameterization is not possible in the case of exceptions relating to subprogram parameters.

IMPORTANCE: IMPORTANT

If Ada is to be used for object-oriented parameterized components.

CURRENT WORKAROUNDS:

The simplest workaround is to avoid the use of exceptions - resulting in the loss of a powerful language feature.

The more usual workaround is to declare the exception that needs to be parameterized in a library package and require that all users of the generic unit supply actual subprograms which raise that exception. For example:

```
package VECTOR_EXCEPTIONS is
```

```
   LENGTH_ERROR : exception;
   -- etc.
end VECTOR_EXCEPTIONS;
generic
   type VECTOR is private;
   function "+" (V,W:VECTOR) return VECTOR;
   -- should raise VECTOR_EXCEPTIONS.LENGTH_ERROR
package USING_VECTORS is
   --etc.
end USING_VECTORS;
```

This relies on close cooperation between the supplier and user of the generic unit, or on the writing of exception translation code by the user of the unit.

POSSIBLE SOLUTIONS:

The proposed solution is to extend the Ada language to permit generic formal exceptions. The proposed addition to the syntax is:

generic_parameter_declaration ::=exception_declaration

As an example:

```
generic
   type VECTOR is private;
   function "+" (V,W : VECTOR) return VECTOR;
   LENGTH_ERROR : exception;
package USING_VECTORS is
   --etc.
end USING_VECTORS;
```

INABILITY TO USE GENERIC EXCEPTIONS

DATE: September 13, 1989

NAME: Jeff Loh

ADDRESS: Intelligent Choice, Inc.
2200 Pacific Coast Highway, Suite 201
Hermosa Beach, California 90254

TELEPHONE: (213) 376-0993

ANSI/MIL-STD-1815A REFERENCE: 12.1

PROBLEM:

Inability to use generic exceptions when building reusable generic units. Consider the following example:

```
generic

        type Objects is limited private;

        with procedure Q(Source : in out Objects;
                         Target : Objects);

procedure P;

procedure P is
begin
 Q;              -- EXCEPTIONS IN Q IS NOT KNOWN AND
end P;           -- CANNOT BE HANDLED PROPERLY
```

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

If there exists only one exception in Q, then the use of "others" will be sufficient. If more than one exception in Q is to be handled, then there is no work around.

POSSIBLE SOLUTIONS:

Modify the definition of generic specification to include exception declarations so that the following can be declared and used.

```
generic

        type Objects is limited private;

        E1 : exception;
```

```
        E2 : exception;

        with procedure Q        (       Source : in out Objects;
                                        Target : Objects);


procedure P;


procedure P is
begin
 Q;

exception
        when E1         => ErrorHandler1;
        when E2         => ErrorHandler2;
        when others     => Panic;
end P;
```

DESCRIBING OBJECTS DISTRIBUTED ACROSS

DATE: 26 September 1989

NAME: Jeffrey R. Carter

ADDRESS: Martin Marietta Astronautics Group
MS L0330
P.O. Box 179
Denver, CO 80201

TELEPHONE: (303) 971-4850
(303) 971-6817

ANSI/MIL-STD-1815A REFERENCE: 12.1

PROBLEM:

Generic Formal Entry Parameters

Task entries can be used as actual parameters for generic formal procedure parameters, but the generic unit cannot make timed or conditional calls to these entries because they are known to the generic unit as procedures. It would be useful to have generic formal entry parameters to allow generic units to make timed and conditional entry calls.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

A generic formal subprogram parameter may be defined with a parameter of type duration or boolean which tells the subprogram to make the correct kind of entry call. This places the conditional or timed entry call in the subprogram, so the generic unit is making the entry call indirectly.

POSSIBLE SOLUTIONS:

Modify the definition of a generic_parameter_declaration in 12.1 to include

```
with entry_specification;
```

for example,

```
with entry add_reading (reading : in reading_id);
```

The matching rules for a formal entry should allow it to be matched by a task entry or a procedure with the same parameter profile. A formal entry instantiated by a procedure will always immediately respond to a call, so any "else" or "or" branches of conditional or timed entry calls to the formal entry will never be taken.

GENERIC FORMAL TASK TYPES

DATE: October 20, 1989

NAME: Thomas J. Quiggle

ADDRESS: Telesoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (619) 457-2700 ex. 158

ANSI/MIL-STD-1815A REFERENCE: 12.1, 12.1.1, 12.1.2

PROBLEM:

When constructing reusable, concurrent components it is often useful to declare a generic unit with a formal limited type, expecting that instantiations of the generic will provide a task type as the actual. If the generic is instantiated with a other than a task type, the intended semantics of the generic unit may not be obtained. Additionally, by failing to make a distinction between task types and other limited types, operations unique to task types (e.g. the TERMINATED attribute and abort statements) are not available from within the generic unit.

Examples of such generic units include ancillary run time packages that implement operations on tasks objects in addition to those defined in the language standard. For example, a distributed Ada application may require global name services for Ada tasks, where names for any other object would be meaningless. The following (simplistic) example illustrates such a name service package:

```
package GLOBAL_NAME_SERVER is

        type NAME is private;

        generic
                type TASK_TYPE is LIMITED_PRIVATE;
        function GENERATE_NAME(T : TASK_TYPE) return NAME;

        -- operations on names defined here

private
        type NAME is ...;
end GLOBAL_NAME_SERVER;
```

IMPORTANCE: IMPORTANT

Generic units with formal limited types that are intended to be instantiated with task types as actuals are subject to inadvertent instantiation with other types.

CURRENT WORKAROUNDS:

Authors of generic units cannot guarantee that such units will not be instantiated with unintended actuals.

POSSIBLE SOLUTIONS:

In section 12.1, extend the definition of a generic parameter to read:

```
generic_parameter_definition ::=
        identifier_list : [in [out]] type_mark [:= expression];
        | type identifier is generic_type_definition
        | task type identifier;                          -- Added
        | private_type_declaration
        | with subprogram_specification [is name];
        | with subprogram_specification [is <>];
```

GENERIC FORMAL ENTRIES

DATE: October 20, 1989

NAME: Thomas J. Quiggle

ADDRESS: Telesoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (619) 457-2700 ex. 158

ANSI/MIL-STD-1815A REFERENCE: 12.1, 9.5

PROBLEM:

When constructing reusable, concurrent components it is often useful to declare a generic unit with a formal subprogram, expecting that instantiations of the generic will provide an entry as the actual parameter. If the generic is instantiated with a procedure as the actual subprogram, the intended semantics of the generic unit may not be obtained. Additionally, by using a subprogram declaration for the formal, conditional and timed entry calls cannot be made from within the generic unit to an entry used as an actual in a generic instantiation.

For example consider the following generic unit (adapted from the CIFO[1] entry on "Asynchronous Entry Calls"):

```
generic
        type PARAMETER_TYPE is private;
        with procedure ENTRY_TO_BE_CALLED(PARAM : in PARAMETER_TYPE);
package ASYNCHRONOUS_ENTRY_CALLS is

        procedure CALL_ASYNCHRONOUSLY(PARAM : in PARAMETER_TYPE);

end ASYNCHRONOUS_ENTRY_CALLS;

package body ASYNCHRONOUS_ENTRY_CALLS is

        task type MESSENGER is
                entry BUFFER_DATA(DATA : in PARAMETER_TYPE);
        end MESSENGER;

        type ACCESS_MESSENGER is access MESSENGER;
                -- The master of all messenger tasks will be the
                -- instantiator this of generic.

        task body MESSENGER is
                BUFFER : PARAMETER_TYPE;
        begin
                accept BUFFER_DATA(DATA : in PARAMETER_TYPE) do
                        BUFFER := DATA;
```

```
            end BUFFER_DATA;
            ENTRY_TO_BE_CALLED(BUFFER);
        end MESSENGER;

        procedure CALL_ASYNCHRONOUSLY(PARAM : in PARAMETER_TYPE) is
            LOCAL_MESSENGER : ACCESS_MESSENGER := new MESSENGER;
        begin
            LOCAL_MESSENGER.BUFFER_DATA(PARAM);
        end CALL_ASYNCHRONOUSLY;

    end ASYNCHRONOUS_ENTRY_CALLS;
```

In the above example, the package Asynchronous_Entry_Calls is intended to be instantiated with an entry as the actual parameter for the formal subprogram Entry_To_Be_Called. Calls to the procedure Call_Asynchronously will create an independent task to rendezvous with the task owning Entry_To_Be_Called, thus allowing the caller of Call_Asynchronously to continue executing even if the call to with Entry_To_Be_Called cannot be immediately accepted. The following sample instantiation illustrates the intended usage.

```
task PRODUCER;

task CONSUMER is
    entry CONSUME_DATA(DATA : SOME_TYPE);
end CONSUMER;

package SEND_DATA is new ASYNCHRONOUS_ENTRY_CALLS
    ( PARAMETER_TYPE     => SOME_TYPE,
    ENTRY_TO_BE_CALLED => CONSUMER.CONSUME_DATA );

procedure ASYNCHRONOUS_SEND(DATA : SOME_TYPE)
    renames SEND_DATA.CALL_ASYNCHRONOUSLY;

task body PRODUCER is
    PRODUCED_DATA : SOME_TYPE;
begin
    loop
        ... PRODUCE_DATA
        ASYNCHRONOUS_SEND(PRODUCED_DATA);
        ...
    end loop;
end PRODUCER;

task body CONSUMER is
begin
    ...
end CONSUMER;
```

IMPORTANCE: IMPORTANT

Generic units with formal subprogram types that are intended to be instantiated with entries as actuals, are subject to inadvertent instantiation with procedures instead.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

In section 12.1, extend the definition of a generic parameter declaration to read:

```
generic_parameter_declaration ::=
        identifier_list : [in [out]] type_mark [:= expression];
    | type identifier is generic_type_definition;
    | private_type_declaration
    | with subprogram_specification [is name];
    | with subprogram_specification [is <>];
    | with entry_specification [is name];          -- added
    | with entry_specification [is <>];           -- added
```

where entry_specification is defined as:

```
        entry_specification ::=
                entry identifier [formal_part];
```

A formal entry and an actual entry would be matched only if both have the same parameter profile. Matching rules for default entries specified by a name, or by a box, follow those for default subprograms. The above definition of entry_specification (as opposed to entry_declaration as defined in 9.5:1), explicitly disallows generic formal entry families.

The matching rules for generic formal subprograms could remain unchanged (i.e. allowing entries as actuals for generic formal subprograms), thus permitting upward compatibility with Ada83.

REFERENCES

[1] "A Catalogue of Interface Features and Options for the Ada Runtime Environment: Release 2.0," ACM SIGAda, Ada Runtime Environment Working Group, December 1987.

GENERIC RECORD COMPONENTS

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 12.1, 12.1.1, 12.3

PROBLEM:

Ada does not currently provide for a generic parameter declaration for record components. Records in generic declarations must be treated as a private type and handled as a whole.

IMPORTANCE: IMPORTANT

This feature is very important for the development of reusable generic components. (Possibly ESSENTIAL)

CURRENT WORKAROUNDS:

Developers of reusable generic components must give clear instruction to users via comments on preparation of selector functions to read and store individual components within records.

POSSIBLE SOLUTIONS:

Add a record type definition similar to those defined in 3.7 to the generic type definition list. Do not require actual record type to match the generic record declaration component for component.

- Match the formal generic record type declaration with the same rules currently existing for formal generic private types.
- Require an actual parameter for each declared component in addition to the actual parameter for the record type.
- Require that all of the components of the generic record declaration be matched by components from the actual record type used to match the generic record declaration.
- Match each formal generic record component with a component having matching type requirements.
- Do not allow one component of the actual record to satisfy more than on component of the generic record declaration.
- Do not require the same order of components.

- Do not require all components of the actual record type to have corresponding generic record components.

- An generic incomplete type declaration for a record must be completed within the generic declaration area.

- An incomplete type should not be matched. Only its completion should be matched.

DECLARATION EXAMPLE:

```
generic

        type Node;              -- no match required

        type Link is            -- match Link with pointer type
                access Node;

        type Node is            -- match Node with record type
                record
                Next:Link;      -- Match Next with component
                end record;

        Default_Node : in Node;

package Controlled_Linked_Records is

        function New_Record
                (Value : in Node := Default_Node)
                        returns Link;

        procedure Free
                (Pointer : in out Link);

end Controlled_Linked_Records;
```

(I would like to see this in the standard library.)

INSTANTIATION EXAMPLE:

```
with Controlled_Linked_Records;
package Some_Tree is
        ...
        type Node_Record;

        type Relationship is
                access Node_Record;

        type Private_Data is private;

        Default_Data : constant Private_Data;
```

```
        type Node_Record is
        record
                Data : Private_Data;
                Father: Relationship;
                First_Son : Relationship;
                Last_Son : Relationship;
                Next_Brother : Relationship;
        end record;

        Empty_Node : constant Node_Record
                := (Default_Data, null, null, null, null);

        package Record_Controls is
                new (Relationship, Node_Record, Father, Empty_Node);
        use Record_Controls;
        ...
private

        type Private_Data is ... ;

        Default_Data : constant Private_Data := ...;
        ...
end Some_Tree;
```

EXCEPTIONS AS GENERIC PARAMETERS

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: 12.1

PROBLEM:

Almost anything that can be named in Ada can be a generic parameter. The three principal items which cannot be generic parameters are generics, packages, and exceptions. Permitting the first two would require significant semantic change, but exceptions have an obvious meaning as generic parameters.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

If an exception is named in the specification of a generic package, it can be renamed in the instantiating package and used for other purposes as well. In addition to being unnatural this is also impossible if the instantiation occurs in the body.

Instead of the exception itself, a procedure can be passed as a generic parameter, and the procedure can simply raise the exception:

```
procedure Raise_Table_Error is
begin
        raise Table_Error;
end raise_table_error;

generic
with procedure Raise_Exception;
package Framework is
...
raise_exception;
...
end framework;

package Instance is new framework (raise_table_error);
```

This is equivalent to what is needed, but clumsy, and potentially inefficient.

POSSIBLE SOLUTIONS:

Permit a generic parameter of the form:

<name> : **exception;**

METATYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 12.1(2)

PROBLEM:

General formal types are not actually "types". They are, more precisely, metatypes; this should be reflected in the standard.

IMPORTANCE: IMPORTANT

This revision request is motivated primarily by concerns about symmetry and aesthetics, but it also makes it somewhat easier to teach generics because the terminology used for generic formal types is more precise and accurate; it also helps make the language more consistent with the terminology used by other modern object-oriented languages.

CURRENT WORKAROUNDS:

Some diligent programmers comment their generic formal types with the comment -- Metatype.

POSSIBLE SOLUTIONS:

Add the METATYPE reserved word to the language, and permit its use in place of the reserved word TYPE in the generic formals region of generic specifications:

```
generic
        metatype Item is private;
package Stack_Generic is...
```

COMPATIBILITY:

The proposed solution is non-upward-compatible. Successful recompilation of previously-compiled code is not guaranteed, because a new reserved word METATYPE has been added to the language, and this word may already be in use in some code (although this is not all that likely).

ALLOW DEFAULT NAMES FOR ALL GENERIC FORMAL PARAMETERS

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 12.1(2)

PROBLEM:

The problem is a lack of symmetry in the generic formal part. A "with subprogram_specification" was provided that has two options for defaults: [is name] and [is <>]. Unfortunately, no defaults were provided for the type name in: **type** INTEGER **is range** <>;

Even though 99% of the time, the name of the generic actual parameter will be INTEGER, there is presently no way to get a default.

The problem causes one of several bad situations.

a) Types that should be generic formal parameters are hard coded in order to save the user from seemingly redundant specification of many generic actual parameters.
b) Users are burdened with long, error prone, generic instantiations.

The request is for a syntax that allows all items in the generic formal part to have at least the default of their own name. The standard named association rules and standard default parameter rules should continue to apply.

```
--      Quantitative Technology Corporation's (QTC) Math Advantage library
--      standard generic formal part :
generic
        type Index_Type is range <>;
        type Integer_Type is range <>;
        type Integer_Vector_Type is array(Index_Type range <>) of Integer_Type;
        type Real_Type is digits <>;
        type Real_Vector_Type is array(Index_Type range <>) of   Real_Type;
        type Real_Matrix_Type is array(Index_Type range <>,Index_Type range <>) of Real_Type;
        type Complex_Type is private;
        type Complex_Vector_Type is array(Index_Type range <>) of   Complex_Type;
        type Complex_Matrix_Type is array(Index_Type range <>,Index_Type range <>) of Complex_Type;

        with function Extract_Real_Part(Complex_Input: in Complex_Type) return Real_Type is <>;
        with function Extract_Imaginary_Part(Complex_Input: in Complex_Type) return Real_Type is <>;
        with function Construct_Complex(Real_part: in Real_Type; Imaginary_Part: in Real_Type) return
                                                Complex_Type is <>;
```

```
package QTC_sample is
end;
--      of the 12 possible generic actual parameters, 9 must be provided,
--      while the last three may be defaults.
--
--      If a syntax was available for default names, a possible instantiation
--      might be :

with QTC_sample;
procedure Y is -- assuming default notation for other generic formal parameters

        package MY_QTC is new QTC_sample( Index_Type =>         INTEGER,
                                                                Integer_Type => INTEGER,
                                                                Real_Type => LONG_FLOAT);
begin
        null;
end Y;

-- note the compromise that all arrays and matrices were assumed
-- to have the same type for an array index. Each could have been
-- it's own generic formal parameter. This would have been a heavier
-- burden on every user instantiation.
```

A similar argument exists for The Numerical Algorithms Group (NAG) generic package:

```
generic
  type FLOAT_TYPE is digits <>;
  type FLOAT_VECTOR_TYPE is array(INTEGER range <>) of FLOAT_TYPE;
  type FLOAT_MATRIX_TYPE is array(INTEGER range <>,INTEGER range <>) of FLOAT_TYPE;
  type COMPLEX_TYPE is private;
  type COMPLEX_VECTOR_TYPE is array(INTEGER range <>) of COMPLEX_TYPE;
  type COMPLEX_MATRIX_TYPE is array(INTEGER range <>,INTEGER range <>) of
                                        COMPLEX_TYPE;
  COMPLEX_ZERO : in COMPLEX_TYPE;
  with function RE(X:COMPLEX_TYPE) return FLOAT_TYPE is <>;
  with function IM(X:COMPLEX_TYPE) return FLOAT_TYPE is <>;
  with procedure SET_REAL_PART(X: in out COMPLEX_TYPE;
                               REAL_PART: in FLOAT_TYPE) is <>;
  with procedure SET_IMAG_PART(X: in out COMPLEX_TYPE;
                               IMAG_PART: in FLOAT_TYPE) is <>;
  with function COMPOSE_FROM_CARTESIAN(REAL_PART, IMAG_PART: FLOAT_TYPE)
                                        return COMPLEX_TYPE is <>;
  with function ARGUMENT(X:COMPLEX_TYPE) return FLOAT_TYPE is <>;
  with function ARGUMENT(X:COMPLEX_TYPE;
                         CYCLE:FLOAT_TYPE) return FLOAT_TYPE is <>;
  with function MODULUS(X:COMPLEX_TYPE) return FLOAT_TYPE is <>;
  -- ...
package GENERIC_REAL_COMPLEX_OPERATIONS is
end; -- six types and one constant must be used as actual generic
     -- formal parameters. Note that the index type is almost
     -- always INTEGER, thus it is not passed as a generic formal
```

```
-- parameter. Even though there are packages that export all
-- the needed types and functions, the user must still write
-- many lines, just to make a one-to-one correspondence
-- between two packages in the same library. Typically there
-- are 20 to 150 packages in a full library. Far too many
-- packages to lump into one giant package.
```

IMPORTANCE: ESSENTIAL

Either the library package implementor penalizes the user by forcing a long list of generic actual parameters, or the library package implementor takes away flexibility some users may need.

CURRENT WORKAROUNDS:

Forced compromise.

POSSIBLE SOLUTIONS:

Pick a simple syntax for defaults. Keep all existing association rules the same.

ALLOW RECORDS IN GENERIC FORMAL PART

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 12.1(2) 12.3(2)

PROBLEM:

There is a lack of uniformity by not having record types in the generic formal part. Arrays and other types may appear in the generic formal part and may be used in generic bodies. Records need to be added so that generic bodies operating on records will be more readable and understandable.

IMPORTANCE: IMPORTANT

When users create numeric types, the structure is often a record. The later use by others of these numeric types needs to have record component information. This requires a record type to be a generic formal parameter as will as a generic actual parameter.

CURRENT WORKAROUNDS:

A private type must be used in the generic formal part. Constructor and selector functions must be created and passed as generic actual parameters.

POSSIBLE SOLUTIONS:

```
generic
        type Component_1 is range <>;
        type Component_2 is digits <>;
        type Formal_Record is -- given in canonical form (order is important)
          record
                A: Component_1;
                B: Component_2;
          end record;
package P is ...
 -- composition and selection functions may be defined if desired

package body P is
        ...
        R : Formal_Record; -- normal use of record
```

GENERIC BODIES WITH IMPLICIT SPECIFICATION

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 12.1(1)

PROBLEM:

Generic bodies with implicit specification.

When in an implementation one recognizes the possibility to implement several similar functions by a generic function, then one should use a generic function. When the scope of the generic function is only its declarative region, there is not much point in separating the specification and the body of such generic function. In fact, this is even less useful than for non-generic subprograms which can be called before being implemented, since in a single declarative region, generic units cannot be instantiated before the body is given, or execution would raise PROGRAM_ERROR due to incorrect elaboration order dependency.

Now, for non-generic subprograms the language allowed to forget about the specification, such that the subprogram_specification incorporated in the body serves as specification. In fact this is a very useful feature to avoid unnecessary information duplication.

Hence, it is unclear why the same is not allowed for generic subprograms:

```
package body SOMETHING is
        generic
                type...
                with function...;
        function GENERIC_FUNCTION_1 (..) return...is  -- illegal      Ada
        begin
        end GENERIC_FUNCTION;

        generic
                type...
                with function...;
        function GENERIC_FUNCTION_2 (..) return...;
        function GENERIC_FUNCTION_2 (..) return...is  -- illegal      Ada
        begin
        end GENERIC_FUNCTION;
...
end SOMETHING;
```

This example above shows the needless repetition that one often sees in generic subprogram declarations.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

To make this possible one could add a syntax rule defining generic subprogram bodies, and add this rule as alternative in rule proper_body:

```
proper_body::=subprogram_body | package_body| task_body|
generic_subprogram_body   generic_subprogram_body    ::=
generic_formal_part subprogram_body
```

GENERIC SUBPROGRAM

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 12.1(1)

PROBLEM:

When in an implementation one recognizes the possibility to implement several similar functions by a generic function, one should use a generic function. When the scope of the generic function is only its declarative region, there is not much point in separating the specification and the body of such generic function. In fact, this is even less useful than for non-generic subprograms which can be called before being implemented, since in a single declarative region, generic units cannot be instantiated before the body is given, or execution would raise PROGRAM_ERROR due to incorrect elaboration order dependency.

Now, for non-generic subprograms the language allowed to forget about the specification, such that the subprogram_specification incorporated in the body servers as specification. In fact this is a very useful feature to avoid unnecessary information duplication.

Hence, it is unclear why the same is not allowed for generic subprograms:

```
package body SOMETHING is

        generic
                type
                with function ...;
        function GENERIC_FUNCTION_1(...)return...is  --illegal Ada
        begin
        end GENERIC_FUNCTION;

        generic
                type
                with function ...;
        function GENERIC_FUNCTION_2(...)return ...;
        function GENERIC_FUNCTION_2(...)return ...is --legal Ada
        begin
        end GENERIC_FUNCTION;
...
end SOMETHING;
```

The example above shows the needless repetition that one often sees in generic subprogram declarations.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

To make this possible one could add a syntax rule defining generic subprogram bodies, and add this rule as alternative in rule proper_body:

```
proper_body ::=
        subprogram_body|
        package_body|
        task_body|
        generic_subprogram_body

generic_subprogram_body ::=
        generic_formal_part subprogram_body
```

STRICTER CHECKING OF MATCHING CONSTRAINTS FOR INSTANTIATIONS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 12.1.1(4-5), 12.1.3(5)

PROBLEM:

The present definition of Ada states that the subtype specified for a generic formal object of mode in out is ignores (12.1.1(4-5)), as is the subtype specified for a parameter of a generic subprogram (12.1.3(5)). We will argue that this is a mistake that should be corrected. We will show how readability and maintainability of source code, quality of generated code, and clarity of the Ada language definition are all affected adversely. Finally, we will propose an alternative that addresses these problems.

Readability and Maintainability of Source Code

It is fairly obvious how these rules can result in confusing program behavior. When, for example, a user calls a function which is specified as taking a parameter of a given subtype, he expects that constraint_Error will be raised if and only if the corresponding actual parameter fails to satisfy the constraints of the given subtype. This would seem to be a logical consequence of the contract model, the model underlying the separation of specification from implementation which is one of the most fundamental concepts of the Ada language design. The language designers recognized this potential source of confusion and deemed it to be so serious that they took the unusual step of including informal recommendations of methods for avoiding legal-but-confusing constructs (12.1.1(5) and 12.1.3(5)). This deviation from the contract model seems particularly inconsistent in light of Ada's strict enforcement of the contract model for formal array and access types (12.3.4.(5), 12.3.5(2)).

One might argue that the present policy is actually quite consistent because it treats generic formal subprograms and in-out mode objects in the same way as renaming declarations (8.5(4,8)). This ignores the fundamental difference that a renaming declaration can always be looked through (both by the user and the compiler) to get information about the renamed object or subprogram. The "specification" and the "implementation" of a renaming declaration are not separated, so no contract is needed.

Quality of Generated Code

In the case of and Ada compiler which implements shared-code generics, the current policy makes constraint-check elimination much more difficult in some very common cases involving calls to generic formal subprograms.

In the following example:

```
generic
```

```
        with procedure P (X : in out Natural);
procedure G;

procedure G is
        Nat : Natural   := 0;
begin
        ...
        P (Nat);
        ...
end G;

with G;
pragma elaborate (G);
procedure Instantiate_G is
        procedure Natural_Param (X  : in out Natural) is
        begin
                ...
        end Natural_Param;

        procedure Positive_Param (X : in out Positive) is
        begin
                ...
        end Positive_Param;

        procedure Integer_Param (X  : in out Integer) is
        begin
                ...
        end Integer_Param;

        procedure  No_Checks_Needed
                                is new G (P => Natural_Param);
        procedure Check_Needed_Before_Call
                                is new G (P => Positive_Param);
        procedure Check_Needed_After_Call
                                in new G (P => Integer_Param);
        begin
                ...
        end Instantiate_G;
```

Consider the constraint checking associated with the call to P from within G. If P were not a generic formal subprogram, then the compiler would note that the subtype of the actual parameter matches the subtype of the corresponding formal parameter and would be able to infer that no constraint checks are needed either before of after the call. But because the subtype specified for a parameter of a generic formal subprogram cannot be trusted, the compiler cannot make use of the information available to it at the call site to eliminate these two constraint checks.

It is significant that this cost is incurred even in the case where the actual subprogram given in an instantiation does have the same parameter subtype as the corresponding generic formal subprogram (e.g. the first instantiation in the proceeding example). Because of the possibility that instantiations with now-matching parameter subtypes (e.g. the second two instantiations in the preceding example) might exist, a performance penalty is incurred in the common case, even if no such non-matching instantiations exist.

It may be that this problem has received relatively little attention because it is only a problem for compilers which implement shared-code generics.

Clarity of the Ada Language Definition

When an array aggregate that has an others choice is passed as a parameter in a call to a generic formal subprogram, it is impossible to determine the "subtype of the corresponding formal parameter" (4.3.2(5)) when compiling the generic. This results in a number of problematic constructs.

The following example illustrates some of these:

```
package String_Subtypes is
        subtype S1 is String (1 .. 10);
        subtype S2 is String (2 .. 20);
        subtype Nonstatic is String (S1'First .. S2'Last);
        subtype Unconstrained is String
end String_Subtypes

generic
        with procedure P (X, Y : String_Subtypes.S1);
procedure G;
procedure G is
begin
        P       (X => (others => 'a'),
                 Y  => ('a', others => 'b'));
end G;

procedure P1 (X, Y : String_Subtypes.S2) is
begin
        null;
end;

procedure I1 is new G (P1);

procedure P2 (X, Y : String_Subtypes.Nonstatic) is
begin
        null;
end;

procedure I2 is new G (P2);

procedure P3 (X, Y : String_Subtypes.Unconstrained) is
begin
        null;
end;

procedure I3 is new G (P3);
```

It is clear that the applicable index constraint for the two array aggregates does not come from String_Subtypes.S1 (12.1.3(5)), but there still remain several possible interpretations for this program.

Perhaps the second aggregate, which requires a static applicable index constraint, should be rejected when the generic is compiled; alternatively, perhaps instantiations such as I2 and I3, which do not supply a statically constrained parameter subtype, should be rejected. Perhaps even the first array aggregate should be rejected when the generic is compiled, but this would be another case of penalizing (by denying the use of a useful language construct) the instantiator who adheres to the contract model just because of the possibility that another instantiator might not. If the first aggregate is allowed, then presumably the interpretation of 12.3.2(4) would have to be broadened in order to reject the instantiation I3, which does not supply a constrained parameter subtype.

Deviations form the contract model result in this kind of undesirable interaction between seemingly unrelated language features.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

If a generic unit has a formal parameter of mode in out, the elaboration of a corresponding instantiation checks that any constraints on the parameter are the same for an actual parameter as for the formal parameter. The exception CONSTRAINT_ERROR is raised if this check fails. If a generic unit has a formal subprogram, the elaboration of a corresponding instantiation checks that any constraints on the parameters of the subprogram are the same for the actual subprogram as for the formal subprogram. The exception CONSTRAINT_ERROR is raised if this check fails.

This would mean that the subtype specified for a generic formal object of mode in out and the subtype specified for a parameter of a generic formal subprogram could be trusted inside the generic, and all of the problems described in the preceding sections would vanish. This solution involves no changes to the static semantics of the language, just some new constraint checks to be made at runtime when elaborating the declaration of an instantiation.

This change is not completely upwardly compatible, but in those cases where instantiations would need to be modified, the required changes would typically be straightforward. In some cases, however, the required changes would be awkward. For example, a program containing:

```
generic
        type T is private;
        with function "-" (L, R : T) return T is <>;
package G is
        ...
end G;

package I is new G (Natural);
```

would now raise Constraint_Error. One would solve this problem by introducing an auxiliary function:

```
generic
        type T is private;
        with function "-" (L, R : T) return T is <>;
package G is
        ...
```

```
end G;

function Difference (L, R : Natural) return Natural is
begin
        return L - R;
end Difference;

package I is new G (Natural, Difference);
```

but this seems somewhat cumbersome and might not even produce the desired program behavior. Another solution would be to allow subtype specifications of the form <type_mark>'BASE, at least for parameters of generic formal subprograms. The example then be written as:

```
generic
        type T is private;
        with function "-" (L, R : T'BASE) return T'BASE is          <>;
package G is
        ...
end G;

package I is new G (Natural);
```

but this would involve a substantial, although upward compatible, change to the static semantics of the language.

ONE-PART GENERIC SUBPROGRAM SPECIFICATIONS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 12.1(2)

PROBLEM:

The language permits non-generic subprogram declarations to be specified either in two parts (spec/Body) or just one part (body-only), but generic subprogram declarations must always be specified in two parts; there is an annoying lack of symmetry to this.

IMPORTANCE: IMPORTANT

This revision request is motivated primarily by concerns about symmetry and aesthetics, but it also yields some practical benefit in that it reduces verbosity (by specifying the form of a generic subprogram once instead of twice) and it reduces maintenance overhead (by localizing modification to the form of a generic subprogram to a single location).

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Permit a generic subprogram to be specified either in two parts (spec/body) or in just one part (body only). The syntax for two part specifications would be as it is now. The syntax for one part specification would be similar to one part specification of non-generic subprograms, except that the keyword GENERIC and the generic parameter declaration would precede the keyword PROCEDURE (for generic procedures) and the keyword FUNCTION (for generic functions).

For example:

```
generic
        type Item is private;
function Is_Convertible (This_Item: in Item) return Boolean is
begin
        [statement]
end Is_Convertible;

generic
        type Item is private;
function Convert (This_Item: in out Item) return Boolean is
```

```
begin
        [statement]
end Convert;
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

RECORDS AS GENERIC PARAMETERS, OBJECT ORIENTED PROGRAMMING, TYPE INHERITANCE, REUSABILITY

DATE: October 25, 1989

NAME: Carl Schmiedekamp

ADDRESS: Naval Air Development Center
Code 7033
Warminster, PA 18974

TELEPHONE: (215) 441-1779

ANSI/MIL-STD-1815A REFERENCE: 12.1.2, 3.7

PROBLEM:

Ada provides only minimum support for record types as generic type parameters. If Ada allowed the specification and matching of generic type parameters to any record types which have at least the specified components it would be much easier to build class-subclass type hierarchies and build reusable (i.e. generic) subprograms for related record types.

Ada does not support the hierarchical definition of new data types by expanding the information held as well as the functionality of existing types. The derived type and subtype mechanisms permit the creation of new type names which can store the same or more restricted information compared to the original type.

It is often useful to be able to create a new type like an existing type but with additional components and where the new type can easily use (i.e., inherit) the functionality of the previous type for the common components. This is often needed for Object Oriented Programming and for data representations used in artificial intelligence programs. This capability is also very useful in building reusable software components.

IMPORTANCE: IMPORTANT

The lack of reasonable support for record type parameters in generics or more generally the lack of support for hierarchically defined types makes it difficult to write programs which are object oriented with data types defined using a class-subclass hierarchy. Software component reuse is also limited for record types because of the lack of this support.

CURRENT WORKAROUNDS:

One workaround is to implement the functionality (or "methods") of the type as generic subprograms with the type as a generic parameter and subprogram generic parameters for each of the record components used. This usually means two subprogram parameters for each record component if it is necessary to both read and set the value of the record component. The major problems with the workaround are that the generic specification becomes large and complex, that the bodies of the generic units are complicated by subprogram calls instead of the more readable record component notation and that a set of subprograms have to be defined for each type for which the generic subprogram is to be instantiated.

POSSIBLE SOLUTIONS:

The suggested solution is to allow within generic specifications a new generic type definition which matches record types that contain at least the specified component names and types. The "partial record" type definition would only specify the components needed by the generic unit, any record with at least those components would match the generic parameter. Within the body of the generic the "dot" notation could be used to access the declared record components of variables of the type. A concise syntax for this should be to allow a "with" in a record declaration which would indicate that the partial record would have all the components of the named record type. This "with" could also be allowed in the type declaration of record types to define the subclass types by specifying that the new type will have all the record components of the named type plus the additional components specified in this record declaration. This solution provides a means to meet most of the needs of the class-subclass hierarchy of object-oriented programming. Multiple inheritance can be supported by simply allowing more than one "with" in a record type declaration.

An example of this solution is:

```
        type class_1 is record
        ......
        end record;

generic
type object_type is record
        with class_1;
        end record;
procedure do_something( object : object_type);

.....

        type subclass is record
                with class_1;
                xyz : some_type;
end record;
```

Then the do_something could be instantiated with both class_1 and subclass. Within the body of the generic (do_something) the specified record components would be visible but not any of the components of the records that were not specified in the partial record declaration of the generic procedure specification.

EXECUTION OF A UNIT BY ITS ADDRESS

DATE: September 25, 1989

NAME: Stephen J. Wersan, Ph.D.

ADDRESS: Code 356
NAVWPNCEN
China Lake, CA 93555

TELEPHONE: (619) 939-3120,
Autovon 437-3120
E-mail: WERSAN%356VAX.DECNET@NWC.NAVY.MIL

ANSI/MIL-STD-1815A REFERENCE: 12.1.3

PROBLEM:

JIAWG's proposed Ada 9X changes include the proposal for "Execution of a unit by its address." In essence, what is being asked for is the ability to pass the name of a unit as an argument to another unit. It should be noted that this capability is of importance not only to systems programmers, but also to numeric analysts writing generalized integration procedures. This was conveniently handled in FORTRAN by use of the external declaration.

The indicated difficulties in providing such a capability in Ada are a) How to handle calling sequences and b) How to handle the propagation of exceptions. The purpose of this Revision Request is to suggest an addition to the language that will permit the desired capability while obviating the difficulties.

IMPORTANCE: ESSENTIAL

This is an area of some importance to system programmers. More importantly, it is an area that diehards can point to with some justification in requesting waivers in favor of assembly language. Anything that can be done to minimize such excuses should be given serious consideration.

CURRENT WORKAROUNDS:

I have seen elaborate "Ada-scams" based on the use of Generic Formal Subprograms to meet this need. These proposals are as much a hinderance to the acceptance of Ada as anything else I can think of. There is no straight-forward, wholly-Ada solution. Currently, one must resort to assembly language.

POSSIBLE SOLUTIONS:

What I propose here is a new composite type declaration. It may actually be a formalization of the way some implementations currently handle subprogram declarations, and something like it must certainly figure in types declared when an Ada compiler is written in Ada. For lack of a better term, I am calling this new type a "calling sequence." Except for the parameter modes (in, out, in out), it is remarkably like a record (even down to the syntax for a default value). As an example, suppose the procedure that one wished to pass had the following formal declaration:

```
function MONEY_IMAGE(     AMOUNT          : MONEY
```

```
                    WIDTH          : Natural;
                    WHOLE          : Boolean;
                    DOLLAR_SIGN    : Boolean) return String;
```

Under the proposed revision, one could declare

```
type IMAGE_CALL is
  calling sequence
      AMOUNT          : MONEY;
      WIDTH           : Natural;
      WHOLE           : Boolean;
      DOLLAR_SIGN     : Boolean;
      return          : String;
  end calling sequence;

type MONEY_IMAGE_CALL is access IMAGE_CALL;

type MONEY_IMAGE is invocation to MONEY_IMAGE_CALL;

MY_CALL : MONEY_IMAGE_CALL := new IMAGE_CALL;
```

Subsequently, in a unit wherein these declarations are visible, one could pass the parameter MONEY_IMAGE(MY_CALL):

```
DISPLAY_ASSETS( . . . MONEY_IMAGE( MY_CALL), . . . );
```

the declaration of the procedure DISPLAY_ASSETS would include a formal parameter "MAKE_IMAGE(PTR): Invocation" (always of mode in) at the matching position in its calling sequence definition.

Inside procedure DISPLAY_ASSETS, the actual parameter would replace the formal parameter as shown here:

```
PUT( MAKE_IMAGE( PTR) );
```

The items of the calling sequence can be assigned appropriate values either via an aggregate when the instance is created (new) or set by individual assignment statements using dotted notation. Scalar items of mode 'in' may be assigned values (expressions) of their declared types. Composite items of any mode and scalar items of mode 'out' or 'in out' may be assigned access types pertaining to their declared types.

The called unit needs some means to determine the mode of the individual items in its "virtual calling sequence." For this purpose, this revision posits a predefined enumeration type Invocation_Mode with elements (IN, OUT, IN_OUT) and a new attribute such that

```
MY_CALL.AMOUNT'mode is of this type, and is IN.
```

The important thing to note at this point is that using the proposed revision, all invocation calls may be characterized as having a single parameter which is an access type. Through this access parameter, all items of the virtual calling sequence are available. The mode attribute allows the called unit to determine the mode of a given item in the virtual calling sequence. This answers the problem of how to handle the calling sequence.

The problem of handling the propagation of exceptions boils down to how to pass the called unit a branch address to handle unhandled or relayed exceptions. This can be done, by putting such an address in an undeclared item in the calling sequence structure introduced above, in a manner similar to that pertaining to discriminants of records. The system programmer who is developing the called unit may be given access to this address via an attribute applied directly to the access parameter. Thus, using the construction MY_CALL'exception at an appropriate place would produce a datum of type SYSTEM.ADDRESS.

NO "NULL" CAN BE SPECIFIED AS AN ACTUAL VALUE FOR GENERIC FORMAL SUBPROGRAM PARAMETERS

DATE: August 27, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 12.3

PROBLEM:

It is not possible to specify "null" as an actual (or default) value for generic formal subprogram parameters.

IMPORTANCE:

CONSEQUENCES:

Generic packages which are written for the general case which must ask for many more generic subprogram parameters than will normally ever be used are difficult to utilize.

CURRENT WORKAROUNDS:

Defining a package containing a series of null procedures whose subprogram specifications conform to the generic parameter structure, and having the user with and use this package and then selectively obscure null procedures with "real" procedures immediately prior to performing the instantiation.

POSSIBLE SOLUTIONS:

Allow "null" to be specified as an actual (or default) value for generic formal subprogram parameters.

Example:

```
with (subprogram specification) is <> := null;

-- If no subprogram having the desired specification
--    is found in the user's environment, then the null
--    procedure will be taken as the default procedure.
```

RESTORING THE CONTRACT MODEL

DATE: October 11, 1989

NAME: J G P Barnes (endorsed by Ada (UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 12.3

PROBLEM:

The original intent of the language was that there should be a contract between the writer and user of generic units. The idea that if the user could match the formal parameters correctly then the body would work. That is there would be no hidden dependency of the body on aspects of the actual parameters that were not wholly expressed by formal parameter requirements.

This model was broken by a late revision to the language which permits an unconstrained type to be an actual parameter provided that the type is ot used in the body in situations which would require the type to be constrained (an obvious example is that objects of the type cannot be declared in the body). This proviso is not expressed in the parameter list. This requires additional documentation for the user if nasty surprises are to be avoided.

IMPORTANCE: IMPORTANT

For a number of reasons it is very important that basic language concepts should be strictly held. The present situation is contrary to the whole generic philosophy.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

A notation must be introduced for the formal type to express the requirement that the actual type be constrained (or have defaults in the case of discriminants, see AI-037). Such a notation is not obvious.

A number of other requests for an extension of generic parameters have been submitted; the contract model should be strictly preserved in their satisfaction.

A PRAGMA FOR SPECIFYING A DESIRED CODE GENERATION STRATEGY FOR AN INSTANTIATION

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Boulevard
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 12.3

PROBLEM:

Although subprogram inlining is an implementation-dependent optimization, it was deemed by the language designers to be sufficiently common and of sufficient importance to warrant a pragma, the INLINE pragma, for specifying when code is to be generated inline for a given subprogram call.

It seems that an analogous pragma would be useful for specifying whether or not a given generic instantiation is to share code with other instantiations of the same generic.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

INSTANTIATION OF NESTED GENERICS

DATE: June 20, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 12.3(2)

PROBLEM:

Instantiation of nested generics.

Generic instantiation of nested generic packages is not very programmer friendly.

When a generic unit needs intermediated declarations (declarations that depend on generic parameters), one needs to use N-level generic packages. Example:

```
generic
        type T_INTEGER is range <>;
package G is
        subtype T_POSITIVE is T_INTEGER range 0.. T_INTEGER'LAST;
        generic
        type T_STRING is array (T_POSITIVE range<>) of CHARACTER;
package G2 is
        ...
        end G2;
end G;
```

The problem noticed is the uncomfortable position of the person instantiating such a package, he should write something like:

```
with G;
package P_INTERMEDIATE is new G (INTEGER);
with P_INTERMEDIATE;
package P is new P_INTERMEDIATE.G (STRING);
```

He needs to invent a unique name, different from all other compilation units, and repeat this name 3 times, and afterwards he might not need it any more (as is the case in the example given).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

In the spirit of condensed notations allowed for scalar type definitions, constrained array type definitions etc... one might propose a condensed notation for nested generic instantiation.

If the syntax of generic_instantiation would be slightly modified, this would add significant programmer comfort:

```
generic_instantiation::=
        package identifier is
                new generic_package_name [generic_actual_part];
generic_package_name::=
         generic_package_name [generic_actual_part
generic_package_name]
```

In that case the instantiation above could be written as follows (provided the P_INTERMEDIATE is not needed anywhere):

```
with G;
package P is new G (INTEGER).G2(STRING);
```

which would be converted to

```
with G;
package package_name is new G (INTEGER);
with package_name;
package package_name is new package_name.G (STRING);
```

by the compiler.

GENERIC FORMAL UNCONSTRAINED PRIVATE TYPES

DATE: September 9, 1989

NAME: Gary Dismukes

ADDRESS: TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9891

TELEPHONE: (619) 457-2700 x322

ANSI/MIL-STD-1815A REFERENCE: 12.3.2(4)

PROBLEM:

It is currently possible for a generic instantiation's legality to depend upon certain usages of a generic formal private type within the generic unit. This is undesirable because it violates the visible contract defined by the generic specification. This special case is inconsistent with the language goal of defining the legality of using a program unit strictly based on its visible interface.

Not only does this place a burden on compilers to implement this unusual case of "retroactive" legality checking (possibly having to defer the check until link time for some library models), but it is confusing to users and hurts portability and maintainability since a simple change to a body may result in invalidation of existing (previously legal) instantiations. It should not be necessary for a user of a generic unit to be aware of the contents of a generic body to determine the legality of an instantiation.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Implementors of generic units need to add documentation to generic interfaces with formal private type parameters stating whether instantiations with unconstrained actual types will be legal.

POSSIBLE SOLUTIONS:

Extend the syntax of generic formal private type declarations to allow specification of whether the actual subtype is permitted to be an unconstrained array type or an unconstrained discriminated type without defaults.

Possible syntax:

type <type_name> **is** [limited] **private** [<>];

Specifying a formal private type with a "box" default (<>) would mean that the generic unit cannot contain any uses of the name of the formal type in a context where either a constraint or default discriminants would be required for an array type or a type with discriminants.

In the absence of the default, the name of the type may be (but is not required to be) used in such contexts, as is true in the current language. However, to maintain upward compatibility, it is still necessary

to permit the actual types given for "non-box" private types to be unconstrained array types and unconstrained discriminated types without defaults. Since this would still seem to require the need for the "retroactive" illegality checks that the proposal was intended to eliminate, it is necessary to add one other language change. The elaboration or execution of a construct using a "non-box" private type within a generic in one of the disallowed contexts must raise an exception (e.g., PROGRAM_ERROR) if the associated actual type is an unconstrained type (of the stated kind). Note that such an exception will only occur for programs that would already have been illegal in Ada-83. In most cases a compiler will still be able to give a warning for such cases if desired (via the same mechanism formerly used to report the illegality). The run-time check itself is simple to implement, requiring only that the classification of the actual type be determinable. This is known at compile-time for inlined generic units and at run-time if the unit is shared.

DEPENDENCIES BETWEEN GENERIC INSTANTIATIONS AND BODIES

DATE: June 26, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 12.3.2(4)

PROBLEM:

Dependencies between generic instantiations and bodies.

Use of formal private types in generic bodies determines whether actuals match or not.

THis rule looks more like a late patch than anything else. In fact, the Rationale published by Alsys sounds a lot different, there it is stressed that the legality checks of specification and body should be done independently, and that verification between instantiation and body should be split int a separate verification between instantiation and declaration on one hand, and a check between declaration and body on the other, exactly as is done with packages.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Replace the patch solution by a proper solution : specify whether formal private types may be used in contexts where only constrained types or mutant types are allowed. One should invent a name for such types (e.g. allocatable, complete, discriminated, bounded, or whatever) and allow a keyword expressing this property in a generic private type declaration. So, in 12.01(02) one would need to replace:

```
generic_parameter_declaration::=
        ...
        |private_type_declaration
        |...
by:
generic_parameter_declaration::=
        ...
        |type identifier [discriminant_part][complete][limited]
        private
        |...
```

ORDER OF EVALUATION FOR GENERIC ACTUAL PARAMETERS

DATE: October 21, 1989

NAME: Stephen Baird

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3600

ANSI/MIL-STD-1815A REFERENCE: 12.3(17)

PROBLEM:

Generic actual parameters for an instantiation should be evaluated in the order of the declaration of the corresponding formal parameters, not in the order specified in 12.3(17).

In the case of a shared-code generic being implemented in a stack machine, one would like the order of evaluation to correspond to the order in which the shared code for the generic expects to find its arguments on the stack.

The current definition requires the different instantiations of the same generic evaluate their generic actual parameters in different orders. For example:

```
function May_Have_Side_Effects (X : Integer)
        return Integer is ...;

generic
        Param 1 : Integer := May_Have_Side_Effects (1);
        Param 2 : Integer := May_Have_Side_Effects (2);
package G is
        ...
end G;
package I_1 is new G (Param_1 =>
                                        May_Have_Side_Effects (3));
package I_2 is new G (Param_2 =>
                                        May_Have_Side_Effects (4));
```

In elaborating the first instantiation, Param_1 must be evaluated first. In elaborating the second instantiation, Param_2 must be evaluated first. This is unfortunate.

It would be better if an implementation at least had the option of using the same parameter evaluation order for all instantiations of a given generic.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

RECURSIVE INSTANTIATIONS

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: 12.3 (18)

PROBLEM:

There are occasions where the ideal solution to the problem would be to write a generic that instantiates itself (perhaps with different actuals) as part of it implementation. Unfortunately, this is not currently permitted by the standard.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Write the first generic as a "skin" over a different, common generic that is instantiated internally. Unfortunately, this is cumbersome.

POSSIBLE SOLUTIONS: Delete paragraph 12.3 (18).

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

For additional references to Section 12. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

0757	DEFINITIONS FOR PROGRAM UNIT AND COMPILATION UNIT	15-37

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 13. REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES

The root of the problem seems to be the definition of unchecked conversion as a generic function. Note that it is not possible to allow general function calls as OUT or IN OUT parameters, but type conversions are different from arbitrary functions, in that they are invertible. In fact, unchecked conversions are especially so, since they should not involve any processing.

This still does not solve the problem of how to block large structures into smaller ones. For this, some explicit form of overlaying seems necessary.

This might be achieved by relaxing the restriction on the use of address clauses to achieve overlaying, and requiring they be supported for general (not just static) address expressions.

Then, for example, the following solution might work:

```
        procedure READ_PARSING_TABLE is
        type TABLE_ALIAS is array(1..TABLE'size/BLOCK'size) of BLOCK;
        ALIAS: TABLE_ALIAS;
      for ALIAS use at TABLE'address;
    TABLE_FILE: BLOCK_IO.FILE_TYPE;
  begin
    .... BLOCK_IO.READ(TABLE_FILE,ALIAS(I)); ....
  end READ_PARSING_TABLE;
```

Since I don't like using global variables, I would prefer this be required to work also for OUT and IN OUT parameters, e.g. procedure READ_PARSING_TABLE(TABLE: out PARSING_TABLE) is ....

```
ALIAS: TABLE_ALIAS;
  for ALIAS use at TABLE'address;
....
```

This feature would probably eliminate the need for unchecked conversions.

A disadvantage is that there is no longer clear visibility that something unsafe is going on (as via "with UNCHECKED_CONVERSION"). This weakness could be corrected by tying this feature to some library unit, just as using 'ADDRESS currently requires package SYSTEM. [ada9x]

ADDRESSING AN DEREFERENCES

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 13

PROBLEM:

Many languages support the safe use of addresses within the application without allowing the programmer to compute address values. Ada has obfuscated addresses so much that the embedded systems have to go to great lengths to interface with the architecture for items that are simple. As a result, the object images cannot be guaranteed to match hardware without rep specs. and other techniques. Surely, there is a better way.

An access type should not point to some intervening structure but should provide the access to the object itself. This will make the need for standardizing the internal compiler interfaces much less a problem. To require that a dereference process always reach the object and not an intervening structure, then Ada can better interface with other languages. At present, all this is implementation dependent so nothing is portable.

IMPORTANCE: VERY HIGH

CURRENT WORKAROUNDS:

guise--unchecked conversions, assembler

POSSIBLE SOLUTIONS:

1. better support for double buffering where the hardware Direct Memory Access (DMA) supports protected read/writes to data and automatic pointer swaps

2. assure the user that a single dereference will find the next object

3. for interfaces to other languages require that the object returned/accessed/referenced be the actual object and not some superstructure, activation record, etc. The programmer can supply any additional information.

STANDARDIZATION OF USER INTERFACE

DATE: October 20, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@SDEVAX.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 13

PROBLEM:

There are many types of interactions with the operating system that occur in a wide variety of programs, especially those involved int he user interface. It would there fore be desirable to have the program determine these attributes using a standard set of calling sequences.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use of system dependent procedures

POSSIBLE SOLUTIONS:

Functions should be provided that will yield the following information.

1. The name of the file that contained that contained the currently running program. This is especially useful because it allows the program to automatically search the directory containing the executable program for the configuration and date files.

2. The name of the file that contained the program that initiated the currently running program. For programs that start other programs by "chaining" or other means, this allows the program to collect additional information.

3. Name of the use running the program. (In some cases this would actually be a part of the system rather than a user.)

4. Parameters that were supplied with the command to start the program.

5. A designation for the terminal being used to run the program.

6. A designation for the type of terminal or workstation being used.

Other useful information would include cursor and mouse information for systems involving those types of

interactions. However, these types of data would be more system dependent.

REPRESENTATION SEPC PLACING SEPCIFIC TASKS IN SPECIFIC NODES IN A COMPUTER NETWORK

DATE: August 18, 1989

NAME: Goran Karlsson

ADDRESS: Bofors Electronics AB
Nettovagen 6
S-175 88 Jarfalla
Sweden

TELEPHONE: +46 758 222 90

ANSI/MIL-STD-1815A REFERENCE: 13 (possibly)

PROBLEM:

In large systems consisting of several computers (typically in a network), there is no way of specifying in which node a task is to be loaded.

IMPORTANCE: ESSENTIAL

For networked systems.

CURRENT WORKAROUNDS:

An inter process communication facility must be written, allowing communication between tasks in different Ada programs possibly running on different computer nodes. The consequence is that each system manufacturer will have his own solution to this problem. This will probably mean portability problems.

POSSIBLE SOLUTIONS:

It would be reasonable to solve this by using representation specifications.

Placing specific tasks in specific nodes:

PLACE task_identifier IN node { node };

node shall be of enumeration type. It should be defined in a globally accessible package such as SYSTEM. Placing the definition of node in SYSTEM is perhaps not wholly acceptable as the definition will be different for different projects.

Problems:

1. What will happen when a rendezvous occurs with a task that has been placed in several nodes?

   The rendezvouz will be with all of the tasks. It will be illegal to use OUT parameters in such a

rendezvouz. (Which task will finally set the OUT parameters?)

2. Where will a task that has no PLACE representation spec be placed?

   It will be placed in all nodes. Any task attempting rendezvous will only rendezvouz with the task in the same node.

The solution to problem 2 will cause a 3rd problem:

3. We will probably want to place some task in just a few nodes, however a rendezvouz will only go to the task within the same node as the caller. That is it will act as a task described in 2, but it will not be placed in all nodes.

   We need to add something to the PLACE representation spec in order to solve this problem.

   PLACE task_identifier IN node { node } LOCALLY;

   This means that the tasks will be placed in the specified nodes, but it will only be possible for tasks within the same node to rendezvouz with them. It will not be possible for a task to rendezvouz with more than one of them. It is possible to have OUT parameters in this type of task.

   PLACE task_identifier IN node { node } GLOBALLY;

   Any task in any node can rendezvouz with this type of task. The rendezvouz will be with all instances of the task. If the task exists in more than one node, OUT parameters are not allowed.

COMMAND LINE PRAGMA

DATE: October 24, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 13

PROBLEM:

Access to command line information is very clumsy and non standard.

IMPORTANCE: IMPORTANT

for the sake of portability

CURRENT WORKAROUNDS:

Where available through a Non standard) *system services package*, access to the unparsed form of the command line is usable. This access service is not very widespread.

POSSIBLE SOLUTIONS:

A pragma should be made standard that specifies that the main procedure parameter list receive the command line parameters as "in" mode parameters.

Example:

```
procedure Parse (File_Name : in String := "") is

    pragma Line_Command_Line_Processor;
    ...
```

The minimum requirement for this feature should be:

- This pragma should be required when the main routine is a procedure.
- The main procedure parameters should be "in" mode parameters with defaulted values for the compilers that do not implement this pragma.
- The pragma should be placed immediately within the declaration area of the main procedure.

-- The converted parameters should be of type Natural of type String.

NO ASYNCHRONOUS EXTERNAL SOFTWARE INTERFACE

DATE: October 20, 1989

NAME: Gene K. Ouye

ADDRESS: Home: 6016 Mardale Ln.
Burke, VA 22015

Work: Attn: Gene K. Ouye, Mailstop TM305
Planning Research Corporation
1500 Planning Research Drive
McLean, VA 22101

TELEPHONE: Home: (703) 451-3267
Work: (703) 556-1838

ANSI/MIL-STD-1815A REFERENCE: 13

PROBLEM:

Section 13.5.1 of the LRM describes a mechanism for associating hardware interrupts with task entry points, however, not all events which could interrupt an Ada program are hardware generated for recognized by hardware. The LRM does not specify any mechanism for accessing and handling any other external events which may occur asynchronously to the execution of an Ada multitasking program. Examples of such events include:

completion of an I/O request,
expiration of a timer,
attempts to communicate with other Ada (or non-Ada) programs, or
depression of a program break key by a terminal user.

What is required is a mechanism to associate task entry points with software generated external events.

IMPORTANCE:

This request is essential. The current Ada language specification is apparently aimed either at embedded systems (i.e., bare machines), where the run-time which is provided with the Ada compiler is the only operating system in the machine and I/O is performed directly via the hardware, or it is aimed at traditional single user applications which only interface to the external world via synchronous file I/O requests. Neither of these scenarios applies to large multi-user applications running on a mainframe with several other users. In such cases, it is necessary to communicate with the external world and to handle expeditiously external events which are generated by the operating system or other programs, but not by the hardware. In order to write such an application, either a vendor-provided system services package must be utilized (which results in an less portable application), or the application must be written in a language other than Ada.

CURRENT WORKAROUNDS: NONE

Unless an external asynchronous event can somehow be mapped to a hardware address (which would allow the use of the for-use at clause described in section 13.5.1), there is no way to handle them in standard Ada.

Some compiler vendors (DEC comes to mind) allow mapping of asynchronous traps to task entries whose behavior is similar to task entries for hardware interrupts.

POSSIBLE SOLUTIONS:

Define an Ada mechanism (such as the pragma AST_ENTRY provided by DEC) which would map task entry points to external asynchronous events. This would at least define a standard Ada method to handle such events, which would promote portability.

REQUIREMENT TO FORMALLY ESTABLISH THE CONCEPT OF VIRTUAL MEMORY IN A DISTRIBUTED/PARALLEL/MULTIPROCESSOR ENVIRONMENT

DATE: August 25, 1989

NAME: V. Ohnjec (Canadian AWG #010)

ADDRESS: 240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 13.x.x, Annex B, Annex C

PROBLEM:

Ada does not currently address the concept of memory management and partitioning in any environment.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Vendor specific run-time systems allow for status data partitioning. The partitioning methods are not necessarily consistent with other vendors, or indeed within the same vendor's different products.

POSSIBLE SOLUTIONS:

Include the concept of run-time system memory management as resident within the validated Ada language. Allow for:

a. Paging
b. Segmentation
c. Various known algorithms

to be standard, but still allow (via pragmas, perhaps) for users to define/use their own/current setup.

This is an add-on enhancement; it should not change current implementations.

REQUIREMENT TO INCLUDE FORMAL MEMORY PROTECTION AND SECURITY TO ADA PROGRAMS IN A DISTRIBUTED/PARALLEL/MULTI-PROCESSOR ENVIRONMENT

DATE: August 25, 1989

NAME: V. Ohnjec (Canadian AWG #011)

ADDRESS: 240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: 13.x.x, Annex B, Annex C

PROBLEM:

Besides data hiding, no method of 'securing' data is available in an Ada program. It should be possible to define 'segments' or 'pages' of memory dynamically throughout the user memory space and allow for these areas to be locked/ unlocked by qualified processes.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Multi-linguistic solutions exist. Nested data hiding remains useful.

POSSIBLE SOLUTIONS:

Allow for the concept of securing data to be a part of the Ada LRM, but as an option. This would ensure minimal negative impact on current implementations.

This is considered as an add-on enhancement to the Ada language.

REPRESENTATION SPECIFICATIONS

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 13.1...13.6

PROBLEM:

For a highly competitive business posture for embedded systems, we have to develop very reliable, efficient software in a very productive manner. Rep. specs are very necessary to applications for embedded systems, but they were added to the language as an afterthought. The current approach in the LRM is fragmented, inefficient. Every format has to have three lines of code written to support the activity.

Representation specs are important to the embedded systems and must be highly efficient both in performance and in compilation speed, code generation efficiency, and coding productivity. Now, the application writer must create several, separate, and fragmented declaration expressions to format data. This adds maintenance overhead and reduces compiler efficiency. It also hampers error recognition. As an optional para., an improvement could provide 15..20% productivity as well as 10..15% in efficiency.

IMPORTANCE:

Very high for any realtime system, in particular for weapons systems

CURRENT WORKAROUNDS:

Generate much Ada code and possible more code to provide assembler packing routines where the compiler doesn't support rep specs. Or, if this is too cumbersome, use a language other than Ada that will allow you to describe interfaces in a better, more precise manner.

POSSIBLE SOLUTIONS:

1. integrate the packing and alignment specifications with the attributes expressed for objects much like other languages, e.g., Cobol, Jovial, Pl-1. These items are further attributes that become part of the symbol table and should not have to be evaluated more than once due to the fragmentation. Further attributes can just be separated by commas and added to the type specification, e.g.,

   name: type, at....., range.....;

   If range is overused and causes parsing problems then use Length specifier. We prefer that "range" is not used to provide the packing control. For multi-record assignments, use the mechanism provided in the language, => for the cases.

2. delete 13.6 as this is available through other means

3. make chapter 13 a requirement with subpara. 13.6, 13.8, and 13.9 as still an option.

REPRESENTATION CLAUSE FOR ARRAY TYPES

DATE: September 27, 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraves AG
8052 Zuerich
Switzerland

TELEPHONE: (011 41) 1 305 33 17

ANSI/MIL-STD-1815A REFERENCE: 13.1

PROBLEM:

The storage representation of arrays cannot be controlled easily, using the available representation clauses. It is e.g. not possible to specify:

- The exact number of bits to be reserved for every component of an array type.
- The alignment of the entire array.

A workaround exists, but is clumsy and reduces the understandability of the source.

IMPORTANCE: IMPORTANT

Representation clauses exist for record types and enumeration types buy not for array types. This seems to be a fault of the language definition which should be removed.

CURRENT WORKAROUNDS:

Two auxiliary record types, one enclosing the array type and one enclosing the component type must be defined. The exact representation of the array type is then controlled with record representation clauses.

POSSIBLE SOLUTIONS:

STORAGE SIZE SPECIFICATION FOR OBJECTS

DATE: October 12, 1989

NAME: John Pittman

ADDRESS: Chrysler Technologies Airborne Systems
MS 2640
P.O. Box 830767
Richardson, TX 75083-0767

TELEPHONE: (214) 907-6600

ANSI/MIL-STD-1815A REFERENCE: 13.1 (10), 13.2a(5), B-2(9)

PROBLEM:

An unaccepted length clause for a type is treated as an error.

IMPORTANCE: IMPORTANT

A common use of a length clause is to specify how much storage a type requires; this generates machine dependent code even though what is required is memory conservation.

CURRENT WORKAROUNDS:

Use different code on different machines.

POSSIBLE SOLUTIONS:

Extend pragma PACK to apply to objects.

USE OF VARIOUS (OPTIONAL) FEATURES, E.G., LENGTH CLAUSES

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 13.2

PROBLEM:

Ada currently supports the use of various (optional) features in the source code to control the build process, e.g. length clauses to control task stack sizes. This is obviously better than no standard at all in these areas, but experience has shown that it is much more convenient if such information, which tends to be compiler and configuration dependent, is kept separate from the source code, as it is then easier to get a system working on an embedded target, and easier to move it from one target to another. Note that it will still be necessary to be able to specify such information at compile time, to a finer level of detail than currently supported, e.g., per object of a task type.

IMPORTANCE: IMPORTANT

Programmers will continue to have to make changes all over their source code when changing compiler or target, with corresponding strain on Configuration Management.

CURRENT WORKAROUNDS:

Non standard build control languages chosen by compiler vendors. Use of task types avoided.

POSSIBLE SOLUTIONS:

LENGTH CLAUSE T'SIZE

DATE: September 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraves AG
8052 Zuerich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 13.2(4-6)

PROBLEM:

FOR t 'SIZE USE size_expression;

specifies only an upper bound for the number of bits to be allocated to objects of type t. This is not sufficient in many cases. A length clause forcing the implementation to allocate exactly the number of bits specified is required.

IMPORTANCE: ESSENTIAL

The current definition of the length claus 'SIZE is unreliable and misleading. A compiler implementor would e.g., be free to change the number of bits actually allocated for objects of a type referenced in a length clause upon every new release. This makes the length clause 'SIZE, as it is defined today, almost useless.

CURRENT WORKAROUNDS:

The effect of a length clause can also be achieved by a record representation clause:

```
TYPE shell IS
    RECORD
      content : t;
    END RECORD;

FOR shell USE
    RECORD
      content AT O RANGE 0..size_expression-1;
    END RECORD;
FOR shell 'SIZE USE size-expression;
```

ALLOW MULTIPLE VIEWS OF DATA WITHIN RECORD REP CLAUSES

DATE: June 12, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 13.4

PROBLEM:

Embedded computer applications rely heavily on record representation clauses to lay out data within words. This facility is used, for example, to model the representation of data exchanged among processors tied together by buses. Depending upon the design of the entire system, this data may need to be interpreted according to one of several different representations. The information required to determine which representation model to use may or may not be part of the data being transferred. We have no reasonable way to model this situation in Ada for the cases in which the discriminant data is not logically part of the record.

13.4:7 states 'Storage places within a record variant must not overlap, but overlap of the storage for distinct variants is allowed'. This statement restricts us from modeling situations in which the information that determines how the data should be interpreted is not physically part of the data.

IMPORTANCE:

This restriction is a definite drawback to the acceptance of the language. The workarounds either result in unproductive use of storage space or in obscure source code.

CURRENT WORKAROUNDS:

There are two methods that can be used to overcome this too-strict restriction when the information determining how the data should be interpreted is not part of the data. One is to declare records that do in fact have discriminants -- even though doing so represents a mismatch to the physical problem. If the data being interpreted utilizes the full data word being transferred, then the artificial discriminant will occupy a full word by itself. This results in less than optimal use of target MACHINE MEMORY RESOURCES. THE SECOND POSSIBLE WORKAROUND IS TO USE CONVERSIONS, INCLUDING UNCHECKED CONVERSIONS, IN ORDER TO INTERPRET THE DATA. THIS SECOND APPROACH DOESN'T WORK IN ALL CASES, CAN RESULT IN LESS THAN OPTIMAL EXECUTION TIME PERFORMANCE AND OFTEN RESULTS IN DIFFICULT TO READ CODE.

POSSIBLE SOLUTIONS:

One solution would be to ease the restriction that storage places within a record rep spec variant must not overlap. Another solution would be to ease the requirements on the appearance of the discriminant. We

do not presume to know the best way to go, but this is a case where the language does not support appropriate ways to model physical situations.

USING RANGE FOR POSITION REPRESENATION

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 13.4

PROBLEM:

Trying to compute or guess the position in a rep spec for an item is very inconvenient. For example, take the one in the LRM one the first line, range 0..7 occupies first 3 bits (reader must compute it) range 10..11 well that is the 4th not the eighth bit. Then using the Ada syntax/semantics in the usual way does the second item only take on values 10 or 11? No, far away in its object type declaration you can find it is boolean--not fitting with a range specification very well. So, that means in the context of an AT, the RANGE takes a different meaning. Range of 10...11 just doesn't compute to bit positions very well. The applications coder cannot assume that the data will be packed.
Further, why should the applications author want to write things like:

```
for object_thing use
  record at data_buffer_1;
        one at 0*word range 1..1;
        two at 0*word range 2..2;
        ---
        sixteen at 0*word range 16..16;
  end record;
```

why not something clearer--like the following examples:

```
        one at position (0*word, 1) length 1;
              --bits counted left to right

or      one at (0*word,1,1);
```

IMPORTANCE:

Very high that rep specs and range definitions have the same meaning throughout the language. Other languages, even back to Cobol and Jovial, allow better definition of rep specs that vendors have little trouble in supporting.

CURRENT WORKAROUNDS:

You are lucky if the vendor supports rep specs--even when they claim to the result may only be what the compiler would do anyway for the object. The court of last resort is assembler language.

POSSIBLE SOLUTIONS:

1. provide a fully defined type with the capability to assign user attributes within the type definition. (This would take care of other difficulties with the language.)

2. Next, the attributes should be expressible in a singie well-identifiable construct for packing control

3. limit the use of range to mean range and not position. It would be nice if the attribute for position/packing control does not appear like a function as the second or third examples above. For example, an attribute like approach would be as follows:

```
type .....

    one: id , 'position at ( 'word=> 0, 'location=>2,
                                        'length=>2*bytes);
```

RECORD PRESENTATION CLAUSE IS EXCESSIVELY MACHINE-DEPENDENT AND NONINTUITIVE

DATE: September 18, 1989

NAME: Lee W. Lucas

ADDRESS: Naval Weapons Center
Code 31C
China Lake, CA 93555

TELEPHONE: (619) 939-5219

ANSI/MIL-STD-1815A REFERENCE: 13.4.5

PROBLEM:

Numbering of bits, hence ordering of fields within a record for which a representation clause is specified, is machine dependent. This renders record representation clauses much less useful, since the source code is not portable. (See discussion below.) Moreover, on machines that number bits right-to-left, fields within records must be "backwards", which is nonintuitive and error prone.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Current workaround is to study the hardware description and try various representation schemes until something works, which is then nonportable.

POSSIBLE SOLUTIONS:

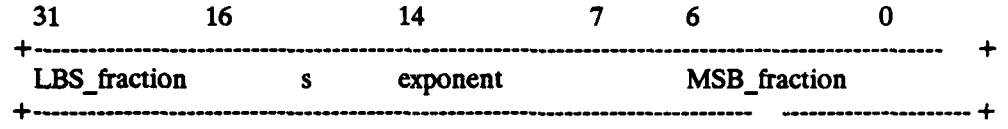
Require all Ada compilers to number storage units and bits within storage units in a consistent manner (left-to-right is preferred) regardless of how the underlying hardware does this numbering. Then the compiler must do the transposition of fields within records and/or bits within fields, if required, not the programmer.

DISCUSSION OF RECORD PRESENTATION PROBLEMS USING VAX Ada

This discussion uses VAX single-precision floating point number representation as an example, because it is easy to confirm that bit patterns in the machine are as expected. Actually, machine floating point format is likely to vary from machine to machine, and so the record representations cannot be portable.

The point of the following discussion is (1) that record representations cannot be written in a portable manner, and (2) that on some machines, for example, VAX, records must be written "backwards", a nonintuitive and error-prone feature.

Single-precision floating point (F_Floating) numbers on the VAX have the following internal representation (bit numbering follows VAX convention).



To most English-speaking people, this suggest the following record structure:

```
type F_Float_Format is
record
        LSB_fraction    : 16-bit-type;
        sign            : 1-bit-type;
        exponent        : 8-bit-type;
        MSB_fraction    : 7-bit-type;
end record;
```

But, the above "natural" record structure is not correct as a representation of F_Floating numbers on a VAX. VAX memory is byte-addressable and bytes are numbered right-to-left in memory. An F_Floating number occupies 4 bytes, with its address being the address of the right-most byte. The value of SYSTEM.STORAGE_UNIT is 8. Since VAX numbers bytes right-to-left and numbers bits right-to-left within bytes, a record representation clause for F_Floating numbers is as follows:

```
type bit        is new Natural range 0..1;
type bits7      is new Natural range 0..127;
type byte       is new Natural range 0..255;
type word       is new Natural range 0..65535;

-- VAX F.Floating format
type F.Float_Format is
record
        MSB_fraction    : bits7;
        exponent        : byte;
        sign            : bit;
        LSB_fraction    : word;
end record;

for F_Float_Format use
record at mod 4;
        MSB_fraction    at 0 range 0..6;
        exponent        at 0 range 7..14;
        sign            at 0 range 15..15;
        LSB_fraction    at 2 range 0..15
end record;
```

Note that the fields of the record are reversed. This can be rationalized as follows: VAX Ada numbers bits "backwards", that is, right-to-left, which means bit patterns within fields of a record will be reversed. One way to get the desire bit pattern is to reverse the fields within the record, and then reverse all bits in the record.

The reversed filed violate psychology (most of us are used to reading and counting from left-to-right) and is nonportable. For certain, there are machines (the 1705A is one) which number bits from left-to-right and for which the record representation clause would have to be written as follows.

```
-- VAX F_Floating format on some other machine
-- won't work on a VAX
type F_Float_Format is
record
        LSB_fraction    : word;
        sing            : bit;
        exponent        : byte;
        MSB_fraction    : bits7;
end record;

for F_Float_Format use
record at mod 4:
        LSB_fraction    at 0 range 0..15;
        sign            at 2 range 0..0;
        exponent        at 2 range 1..8;
        MSB_fraction    at 3 range 1..7;
end record;
```

The purpose of the following remarks is to convince the reader that the above "natural" record representation clause will not work on a VAX. A program was written to view the bit patterns of VAX F_Floating numbers in four different ways (with the help of Unchecked_Conversion). The four different ways were.

(1) Float;

(2) **array** (1..32) **of** bit;

(3) Integer;

(4) F_Float_Format;

The following table shows the values expected (and obtained as program output) for each of these (except the array of bit).

Float	F_Float_Format			Integer		
	LSB	s		exp	MSB	
0.25		0	0	127	0	16256
0.5		0	0	128	0	16384
1.0		0	0	129	0	16512
-1.0		0	1	129	0	49280
129.0		0	1	136	1	17409
-129.0		0	1	136	1	50177
875.0	49152	0	138	90	-1073724070	

The program listing is as follows:

```
with Text_IO;
with Unchecked_Conversion;
```

```
procedure fptest is
        package Integer_Text_IO is
                                new Text_IO.Integer_IO(Integer);
        use Integer_Text_IO;
        package Float_Text_IO is
                                new Text_IO.Float_IO(Float);
        use Float_Text_IO;

        type Bit is new Natural range 0..1;
        type Bits7              is new Natural range 0..127;
        type Byte               is new Natural range 0..255;
        type Word               is new Natural range 0..65535;

        type F_Float_Format is
        record
                MSB_fraction            : Bits7;
                exponent                : Byte;
                sign                    : Bit;
                LSB_fraction            : Word;
        end record;

        for F_Float_Format'SIZE use 32;
        record at mod 4;
                MSB_fraction    at 0 range 0..6;
                exponent                at 0 range 7..14;
                sign                            at 0 range 15..15;
                LSB_fraction    at 2 range 0..15;
        end record;

        type Long_Word is array (1..32) of Bit;
        pragma PACK (Long_Word);

        f : File_type;

        function To_Integer is new Unchecked_Conversion
                                                        (SOURCE => Float,
                                                         TARGET => Integer);

        function To_Long_Word is new Unchecked_ Conversion
                                                        (SOURCE => Float,
                                                         TARGET => Long_Word);

        function To_F_Float is new Unchecked_Conversion
                                                        (SOURCE => Float,
                                                         TARGET => F_Float_Format);

        procedure Print (x : IN Float) is
        begin
                set_col(12);

        put(x, FORE => 4, AFT => w, EXP => 0); put (" ");
```

```
        declare
                w: Long_Word := To-Long-Word(x);
        begin
                for i in reverse w'RANGE loop
                                put(Integer(w)(i)), WIDTH => 1);
                end loop;
        end;
        new_line;    set_col(12);
        declare
        f : F_Float_Format := To_F_Float(x);
        begin
                put(Integer(f.LSB_fraction), WIDTH => 8);
                put(Integer(f.sign), WIDTH => 2);
                put(Integer(f.exponent), WIDTH => 6);
                        put(Integer(f.MSB_fraction), WIDTH => 4);
        end;
end Print;

begin
        Create(f, OUT_FILE, "fptest.out");
        Set - Output (F)
        Print (                0.25);
        Print (                0.5);
        Print (                1.0);
        Print (               -1.0);
        Print (              129.0);
        Print (             -129.0);
        Print (              875.0);
end fptest;
```

A program using the "natural" order in of fields in the record representation of F_Float_Format did not produce the correct output.

RUN TIME INTERRUPT ENTRIES

DATE: August 29, 1989

NAME: William L. Wilder

ADDRESS: Naval Sea Systems Command
Department of the Navy
Washington DC 20352-5101

TELEPHONE NUMBER:

ANSI/MIL-STD-1815A REFERENCE: 13.05

PROBLEM:

It should be possible to create multiple task objects of the same task type for multiple interrupting devices of similar kind. The number of device driver tasks and their binding to interrupts should be determinable at run time.

IMPORTANCE:

RTOS writers will be required to introduce multiple, distinct, task types for each possible interrupting device of the same device and forgo efficient dynamic configuration of systems. The introduction of these different types will make it impossible to pass the resulting task objects as parameters to subprogram or entry calls because of Ada's strong typing rules. Also code implementing the different task types will be replicated, or worse, code for unused device drivers will have to be included in an RTOS to provide for a maximum number of identical devices that seldom exist.

POSSIBLE SOLUTIONS:

It is typical in Run Time Operating Systems (RTOS) to have multiple identical interrupting devices, differentiated in hardware by channel number or some other mechanism that can be cast into an Ada interrupt address. In Ada interrupts are handled by tasks with interrupt entries, and one or more of these tasks are typically used to create device drivers. For multiple identical devices it would be natural to declare multiple objects of the same task type. This cannot be done because the address clause that associates the interrupting device to the task entry is associated with the type of the task rather than the task object. Thus this address clause is only evaluated once when the task type is elaborated and cannot be changed for each task object instance. As a consequence, dynamic configuration of device drivers for an RTOS is difficult or impossible in Ada.

Proposed Solution

One solution is to permit the address clause for a task entry to appear in the task body. Alternatively, the language could provide some other construct which appears in the body of a task unit to bind the task entry with an interrupt source.

Another solution is to provide some form of initialization mechanism for task objects similar to that available for numeric or record objects. This initialization would allow the association of task entries with interrupt sources at task object creation time.

AMBIGUITY IN THE DEFINITION OF ADDRESS CLAUSE

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 13.5

PROBLEM:

Para. 13.5:1 states 'An address clause specifies a required address in storage for an entity.' 13.5:4 states '(a) Name of an object: the address is that required for the object (variable or constant)' We have encountered different interpretations of these two statements from compiler to compiler. The ambiguity reduces to determining who or what is responsible for binding the object to the specified address. Some implementations view the binding as their responsibility; at least one other implementation views the binding as not its responsibility. Apparently both these interpretations are reasonable since all compilers we've dealt with have been validated.

IMPORTANCE:

With the current ambiguity, applications cannot rely on a generally accepted view of the semantics of the ADDRESS clause. The result is that applications that have a legitimate need to use the ADDRESS clause are well-advised to do detective work to verify that their compiler implements the clause in a manner consistent with their needs. This need to experimentally determine the semantics of the language as viewed by a particular implementation (both here and in other instances) is a substantial drawback to the language. Clearing up the ambiguity will decrease the amount of negotiations to be done between implementor and end user and result in a better-defined language.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Change the first sentence in 13.5 to: 'The use of an address clause specifies the address at which the Ada implementation is to allocate the entity.' Change the definitions in 13.5:4-6 similarly. Explicitly state that the use of an address clause does not modify the elaboration process for an object and reference 3.2.1.

THE LRM DOES NOT CONSIDER THE NEED FOR EXECUTING AN ACCEPT ON A HARDWARE INTERRUPT LEVEL

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: 13.5.1

PROBLEM:

The LRM does not fully consider the need for, and implications of, executing an accept on hardware interrupt level. The LRM leaves it open whether the accept will be executed on the hardware interrupt level, or simply at some high software priority. More guidance from the LRM is required, because the current situation is that (apart from some work by ARTEWG) the meaning of interrupts in Ada is almost entirely implementation defined.

IMPORTANCE: ESSENTIAL

Programs that use interrupts will continue to be non-portable, and sometimes be partly written in languages other than Ada.

CURRENT WORKAROUNDS:

A variety of compiler-specific mechanisms, some of which involve polling.

POSSIBLE SOLUTIONS:

The problem arises, I think, because of the differences between processor architectures in the way that they handle interrupts. There are three cases I can think of which are worth distinguishing:

a) Each interrupting device can have its own interrupt level, and tasks can be run on these interrupt levels and can be suspended on entry queues, timer delays etc. (essentially, there is no distinction between hardware and software priority levels).

b) Interrupts are acknowledged and reset automatically by processor logic, but cannot be suspended.

c) Device and/or configuration dependent code must be executed on the hardware interrupt level.

The LRM seems to assume (a), which is not properly supported by any architecture I know about.

Processors in which the priority level is part of the Processor Status Word (PSW) usually correspond to (b). With these processors, it is relatively easy to perform the accept on a software priority (although this may be unacceptably slow).

Processors with external interrupt controllers correspond to (c). It is often necessary to execute non-trivial amounts of code on the interrupt level, so that there is a need to write this code in Ada. The accept therefore has to be written in Ada, and therefore cannot use the full range of Ada features without serious priority inversion occurring.

The various (e.g., ARTEWG) proposals to limit the range of Ada features usable by interrupt accepts have the disadvantage that they involvethe very nastiness (polling instead of multi-tasking) that Ada was meant to make obsolete.

The only remedies that will work on existing hardware are:

1) Change the LRM to make it clear that accepts called by interrupts will not be able to use the full range of Ada features, and may have to use implementation defined features other than a rendezvous to communicate with other tasks.

2) Change the LRM so that the use of interrupt procedures, written in Ada, using asynchronous communications with Ada tasks (via a pre-defined procedure), becomes an optional part of the Ada standard, and the notion of task entries being called by interrupts disappears. This would rationalize something that is already used by several compiler vendors.

Whether or not an Ada interrupt procedure is used, an asynchronous communication mechanism will be needed to ensure that the processor drops off interrupt priority quickly regardless of the state of the tasks, without the polling required by the use of a conditional rendezvous.

DEFINITION OF HARDWARE INTERRUPT HANDLING

DATE: September 28, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

E-mail: ROSS@SDEVAX.GE.COM

TELEPHONE: (215) 337-9805

ANSI/MIL-STD-1815A REFERENCE: 13.5.1

PROBLEM:

Modifications should be made to the language definition for interrupt handling. The current definition uses the same type for memory addresses and interrupt addresses, which potentially poses a restriction on the architecture of the system.

In addition, my reading of the language definition indicates that it is legal to have two entries referring to the same interrupt address.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Since this section is implementation dependent, any workarounds would also be implementation dependent.

POSSIBLE SOLUTIONS:

My first suggestion is to add a new type INTERRUPT_ADDRESS to package SYSTEM. For backward compatibility, this could be defined as a subtype:

```
subtype INTERRUPT_ADDRESS is ADDRESS
```

where applicable. By using its own subtype, the methods of describing interrupts is left completely up to the implementor. If desired, the INTERRUPT_ADDRESS could even be made a record type to increase flexibility for more complex systems. (For example, a system might be designed which would have two different buses for hardware interrupts. The first part of the record would indicate which hardware interrupt system was used, while the second would identify the interrupt.)

I also feel that using the same address for two different interrupt entries should be stated as being erroneous. This would seem to be required, but is not explicitly stated. If it is necessary to start two different tasks from the same hardware interrupt, the interrupt handler can be used to trigger two different tasks by separate calls to the tasks.

INTERRUPT HANDLING, ENTRY ASSOCIATION

DATE: September 27, 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraves AG
8052 Zeurich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: 13.5.1

PROBLEM:

Interrupt handling and interrupt association, as defined by the standard, poses a number of problems. Please consider a few examples:

- Associating an interrupt with an entry is not always the most convenient way to handle the interrupt. It is e.g. difficult to immediately reset the interrupt request (which is required by some hardware), or to make sure that no interrupts are over lost. A task cannot be expected in general to be immediately ready to accept an interrupt.

- The ambiguous use of type system.address for both memory addresses and interrupt (vector) addresses is not appropriate. A separate type, say system.interrupt_address is required. Interrupt addresses sometimes have an internal structure not present in memory addresses. On the other hand, operations usually defined for memory addresses are meaningless for interrupt addresses.

- The association of interrupts with interrupt handling constructs (entries in the current definition of the standard) should be possible by other means than by address clauses. An address clause (as defined in 13.5.1) may only bind the interrupt to an entry of a single task or to all tasks of a task type! It is not possible to associate an entry of a specific task object with an interrupt.

- The entry call semantics (ordinary, timed or conditional) of an interrupt may be highly significant to the functionality of an application program. However, Ada offers no way to specify the required semantics.

The entry call semantics does not only depend on the underlying hardware, but also on the run time system. Programs may thus behave differently on different implementations, even if the hardware is the same.

IMPORTANCE: ESSENTIAL

The current definition of the standard, concerning is far from being sufficient to satisfy the users needs. Implementation-dependent, non-portable features have thus emerged and will certainly be used and elaborated further. This will make Ada programs handling interrupts inherently non-portable, even if they are targeted to the same hardware.

CURRENT WORKAROUNDS:

No general workaround is available. Some implementations provide extended interrupt handling features.

POSSIBLE SOLUTIONS:

Most implementations already provide some (non-portable) extended interrupt support, e.g:

- Allowing subprograms to handle interrupts.
- Pragmas to associate subprograms and task entries with interrupts.
- Pragmas to specify interrupt semantics.

An interesting solution was suggested by Rational, Santa Clara. In their implementation for MC68020, interrupt may be bound to procedures, task entries or a combination thereof, using an implementation-defined pragma.

EFFICIENT AND LESS DANGEROUS WAYS TO BREAK STRON TYPING

DATE: September 6, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634
E-mail : BBardin@ajpo.sei.cmu.edu

TELEPHONE: (714) 732-4575

ANSI/MIL-STD-1815A REFERENCE: 13.5(8), 13.10.2

PROBLEM:

The fundamental issue to be addressed here is how to more efficiently and prudently break the strong typing model of Ada when this is necessary, for instance when non-ascii data enters the program through input or leaves the program through output.

Four techniques are in common use on real projects:

1) unchecked conversion (the intended method),
2) overlays using address clauses,
3) untyped move procedures, and
4) escape to other languages through pragma Interface.

Unchecked conversion is sometimes too inefficient; overlays are defined by the standard as erroneous and they are intrinsically unsafer than unchecked conversion; untyped move procedures rely either upon overlays or on pragma Interface; and use of pragma Interface sidesteps the issue, because it should not be necessary to go outside of Ada in order to break the strong typing of Ada in a controlled and efficient way.

The use of address clauses to achieve overlays of objects is erroneous, although most real-time programmers expect and demand that address clauses support overlaying as a matter of course. Such usage creates aliases for objects which can easily be abused, leading to buggy and unmaintainable programs.

Sometimes overlaying is done in order to "reuse" memory in languages where storage is statically allocated. This is fundamentally a bad practice because an access may inadvertantly be made through an alias after the memory has been re-used for another purpose. However, in Ada, overlaying in order to reuse memory is unnecessary because the desired effect can be achieved safely (and automatically) by placing the declarations in a dynamic scope with the appropriate lifetime.

Overlaying may also be used in order to break the strong typing of Ada at the external interfaces of the program. Unchecked conversion is the language feature which was designed to do this, but in actual use, although the conversion itself may usually be implemented at zero cost, the total cost of breaking the typing model is frequently larger than if overlays are used.

If the size of an overlay is commensurate with the size of the object being overlaid, e.g., a single component of an array is overlaid in its entirety, then techniques requiring unchecked conversion may be sufficiently efficient that they may be used in practice. Even in this case, however, the amount of code that must be written in order to use unchecked conversion is greater than that needed to overlay using an address clause, so that programmers may still prefer to use the latter.

However, there are many applications which must deal with externally specified messages (data) in multiple formats which must be parsed or scanned sequentially in order to interpret them. That is to say, the interpretation of one part of the data is dependent on the interpretation of some previously interpreted part or parts. Conversely, on output, a message must often be built up incrementally, with its location in the output buffer being variable and data driven. In either case, it is necessary to view arbitrarily positioned slices of a message input buffer as different strong types.

Under these circumstances, an approach using unchecked conversion is intrinsically expensive due to the basic costs of implementing slices and the number of checks required by Ada semantics. Because of this, real-time or performance-critical projects tend to use overlays instead, in spite of their philosophical drawbacks, since they are implementable at nearly zero run-time cost.

But overlaying through address clauses is easy to misuse. In particular, the definition of the overlay may be textually separated from its use, and such use is not possible to detect syntactically.

For instance:

```
package P is
        OT : T; -- The object to be overlaid
end P;

with P;
package Q is
        OU : U; -- The overlaying object
        for OU use at P.OT'Address; -- OU is now an alias for OT
end Q;

with Q;
procedure R is
begin
        Q.OU : <some value>; -- OT now has a different value.
        -- OT can be altered in many places in the program through this alias.
        -- There is no local evidence that OT has been changed, which makes
        -- this usage error-prone.
        ...
```

As the situation stands real-time projects will continue to use overlays achieved through the use of address clauses or untyped moves. This practice will, in the long run, lead to large amounts of code which is unnecessarily error-prone and less maintainable than it could be.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Depend upon projects to use less efficient constructs in conjunction with unchecked conversion. In the

most difficult cases this requires slicing, sliding the slice so that a conversion can be performed, and then unchecked conversion to access an object as another type. This technique is tedious to use because it requires many declarations to achieve a single strongly-typed view into the buffer and the code which must be generated to do this can be expensive at best. (The unchecked conversion itself has (or should have) nearly zero run-time cost.)

It is instructive to consider a simplified version of this idiom in a real implementation. (The example given here uses a simple integer type as the target type where, in general, a composite type with a fully specified representation would be required.)

Assume that the width of the desired view is some multiple of the width of an array element. (This is needed when the possible alignments of the view are separated by less than the width of the view.) Then we must perform unchecked conversion on a slice of the array. In order to convert a slice to another type, it is necessary to define a subtype to which the slice may be explicitly converted (slid), and which may be used in instantiating unchecked conversion. Because it is impossible to define a subtype of a constrained array type (which is a (first-named sub)type and therefore has an anonymous base type (see 3.3.2(5))), an unconstrained array type must be used.

```
Word_Size : constant := 16;
Int_Size : constant := 2 * Word_Size;

Max_Words_per_Buffer : constant := 2000;

type Word is range - 2**(Word_Size - 1) .. 2**(Word_Size - 1) - 1;
for Word'Size use Word_Size;
-- The array component type

type Int is range  -2**(Int_Size - 1) .. 2**(Int_Size - 1) -1;
for Int'Size use Int_Size;
-- The type of the overlay

type Index is range 1 .. Max_Words_per_Buffer;

type Buffer_Type is array (Index range <>) of Word;
pragma Pack (Buffer_Type);

subtype Constrained_Buffer_Type is Buffer_Type (Index);
-- Constrained_Buffer_Type'Size must be Word'Size * Max_Words_per_Buffer

          ************ These declarations are necessary for *************
         *********** the uncheckecd conversion approach only ***********

-- A subtype of the unconstrained array type which is the same size as
-- Int is needed.
Last_Slice_Index : constant Index := Int_Size/Word_Size;
subtype Int_Range is Index range 1 .. Last_Slice_Index;
subtype Int_Slice is Buffer_Type (Int_Range);
-- Int_Slice'Size must be Word'Size * Last_Slice_Index
-- A slice the same size as Int

function To_Int is new Unchecked_Conversion (Int_Slice, Int);
```

```
function To_Int_Slice is new Unchecked_Conversion (Int, Int_Slice);

        ********************************************************************

I : Int := <some value>;
B : Constrained_Buffer_Type;

B(N .. N + 1) := To_Int_Slice(I); -- Put I in the buffer at index N

--      The cost of all of this is:
--      unchecked conversion (should be no cost)
--      slice assignment
--      (check constraints on component subtypes,
--      check for matching components,
--      convert subtype (slide),
--      move data)
--      Actual cost of a particular real implementation:
--      ~15 machine instructions.

I := To_Int(Int_Slice(B(N .. N + 1))); -- Get I back

--      The cost of all of this is:
--      slice
--      (check bounds of slice against index,
--      move data?)
--      explicit subtype conversion (slide)
--      unchecked conversion (should be no cost)
--      assignment (move data)
--      Actual cost of a particular real implementation:
--      ~30 machine instructions plus 1 run-time call.
```

The same thing implemented using overlays:

```
I : Int := <some value>;
B : Constrained_Buffer_Type;
...

declare
        J : Int := <some value>;
        for J use at B(N)'Address;
begin
        J := I; -- Put I in the buffer at index N.
        I := J; -- Or, copy the value in Buffer(N) into I.
end;
```

The cost in both of these cases is zero for the conversion itself. The cost of the assignment is just that of moving the data.

In this case, we must depend on the self-discipline of programmers to use locally-defined overlays of limited scope and lifetime, (which is safer than overlays declared in library packages).

A third alternative which is sometimes chosen by real projects is to define a (byte) move procedure, implemented using pragma Interface or in Ada using overlays (and, possibly, pragma Interface).

```
procedure Move(From : System.Address;
               Into : System.Address;
    Number_of_Bytes : Positive);
```

This has the advantage that it need be defined and implemented only once, and the efficiency of the implementation is under the project's control, at least to some degree. Its disadvantage is that it always requires copying (and that copying may be more expensive for small data items than either overlays or unchecked conversion).

Neither unchecked conversion nor overlaying require moving data prior to accessing it (it can be accessed in place, using selection or indexing). Only unchecked conversion requires slicing.

POSSIBLE SOLUTIONS:

Relax the semantic requirements on explicit conversions (4.6) to allow the "explicit conversion" of the address of an object to any type. Only expressions which denote the address of an object to be converted would be allowed, not values of type System.Address (so that global aliases cannot be constructed).

For instance:

```
OT  : T := <some value>;
OTA : System.Address := OT'Address;
OU : U := U(OT'Address);    -- views the contents of OT as a value of
                            -- type U and assigns it to OU (without
                            -- checking if the target has the same
                            -- subtype or underlying representation
                            -- as the converted value)
OU := U(OTA); -- illegal (OTA is a value of type System.Address)
```

If U is a composite type, then referencing subcompontents directly by indexing, selection, or slicing could be performed. So, if U were an array type:

```
type C is ...;
type Index is range 1 .. 10;
type U is array (Index) of C;
U(OT'Address)(2 .. 3)
```

would be a slice. Similarly, if U were a record type with an Integer component, C:

```
type U is
  record
    C : Integer;
  end record;

U(OT'Address).C
```

would be an Integer value.

Note that any object can be a source, including subcomponents of composite objects. The compiler computes the destination address and the amount of data to reference, based on the size of the destination object. The programmer supplies the address of the source of the data.

This approach has four important properties:

1) every "overlay" is locally defined (it cannot be declared anywhere except at the place where it is used),

2) it is immediately recognizable by its syntactic form as an explicit overlay (and all places where it is used can be found by a simple syntactic scan of the source code),

3) it can be implemented at zero run-time cost, and

4) it would be an upward-compatible change to the language.

Property one makes it safer than overlays achieved through address clauses, property two makes it possible to monitor and/or prevent its use, and properties three and four make it acceptable to users

An alternative syntax would be to define an attribute of an object which is itself either universally assignable or which can be explicitly converted to the target type. For example:

```
OU := OT'Representation; -- Uncheckable, except possibly that
                         -- OT'Size >= OU'Size
  OR

OU := U(OT'Representation);  -- At least the intent is expressed
```

A similar, more restrictive, but safer variant of this technique would require that sizes of the objects involved match. In this case the syntax would have to change so that the size would be implied. For example, we might have something like:

```
U(OT(I .. I+3)'Address);
```

where OT is an array object and the slice must be the same size as U. A run-time check would be required on the bounds of the slice, but not on its length, because the length is statically determined (in this case).

Finally, the question of how to assign strong values into a weakly-typed buffer should be addressed. What we need in general is to assign to a slice of the buffer. Perhaps something like this could be used:

```
type Byte is ...;
type Index is range 1 .. 100;
type Unconstrained_T is array (Index range <>) of Byte;
subtype T is Unconstrained_T(1 .. 10);
subtype T_Slice is T(1 .. 2);
OT : T; -- the weakly-typed buffer object

OT(I .. I+1) := T_Slice(OU'Representation);
```

PRIORITIES OF INTERRUPTS

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.,
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.5.1(2)

PROBLEM:

The rule in 13.5.1(2) that interrupts act as entry calls issued by a hardware task whose priority is higher than the priority of any user-defined task is ill-conceived and may have dire consequences in applications.

There is no reason to believe that, in an embedded system, the handling of an arbitrary hardware interrupt should take precedence over currently executing software tasks.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The determination of the priority of the interrupt (rendezvous) vis-a-vis software rendezvous should be left application-dependent.

INTERRUPT HANDLING IN ADA

DATE: November 31, 1989

NAME: Terry Shepard

ADDRESS: Royal Military College of Canada
Kingston, Ontario
K7K 5LO

TELEPHONE: E-mail: Shepard@qucis.queensu.ca

ANSI/MIL-STD-1815A REFERENCE: 13.5

PROBLEM:

Editor's Note: The following Revision Request was submitted to comment on specific problems with Task Type Interrupts (E) and Backlogging Interrupts (F). These problems were identified by the Real-time Embedded Systems Working Group at the Ada 9X Project Requirements Workshop held in May 1989 in Destin, Florida.

These two items are grouped together here because it was felt that they were just symptoms of a larger problem, which is the need to correct a number of aspects related to the handling of interrupts in Ada. There was consensus in the group in support of Task Type Interrupts and in opposition to Backlogging Interrupts. In general, it was agreed that the practical use of the existing interrupt handling facilities was limited, and that changes can be made to make more practical use possible.

SUPPORTING POINTS:

1. Dynamic connection of interrupts to entries, as proposed under E., is a necessary step toward being able to write flexible device drivers that can handle interrupts from sources that are determined at run-time. It is also important to be able to instantiate tasks at compile time that are bound in a flexible way to particular interrupts. This can be accomplished through the use of generic packages that encapsulate parameterized tasks, but the use of a package for this purpose is overkill and should not be necessary. It can also be accomplished by encapsulating the driver task inside another task, but this creates an additional layer of delay and complexity in dealing with interrupts. An encapsulated task also cannot be separately compiled. (LRM (section 13.5) states that the address clause used to bind an interrupt can use a simple expression, meaning that a variable of type SYSTEM.ADDRESS can be used. If this is in fact the case, then dynamic connection of interrupts to entries would appear to be possible already.)

2. The reason backlogging of interrupts was felt to be a bad idea was that in real-time systems, if an interrupt cannot be dealt with at the time, it is unlikely that resources will subsequently become available to deal with it. In other circumstances, in which backlogging does make sense, then at the very least control is required to specify when it is not to be done, so that if there is not enough computing power to handle backlogging and meet the real system needs at the same time, the real system needs can still be met. It is noted that LRM section 13.5.1 explicitly discusses the possibility of queued interrupts, but does not require that interrupts be queued, nor does it require that the user be given control over whether interrupts are queued or not.

3. It was agreed that interrupt entries needed to be protected against calls from other sources, as this can be the cause of serious bugs.

4. If tasks were allowed to schedule other tasks, then interrupt handling stubs (which might be invoked without a full context switch) could be written that could decide to invoke interrupt handling tasks only in particular circumstances. This reduces tasking overhead, and gives finer control over the response to interrupts. It was generally agreed to be a useful idea. The idea of having special "light-weight" tasks that support interrupts was suggested.

5. Asynchronous rendezvous, meaning that both parties need to be there to cause the rendezvous to occur, was felt by some members of the group to be a good idea was well. The problem with asynchronous rendezvous is storage management and cleaning up after tasks that don't complete their rendezvous. This problem is worse in a multi-processor system, especially one in which memory is not shared between the parties engaged in the rendezvous.

6. The mechanism by which access to an interrupt handler is specified needs revision. Giving an address is inappropriate in some architectures. For example, an interrupt "address" may be a vector number, which cannot be jumped to. There may be tables of vector numbers, so the table may need to be identified as well. On computers that support virtual machines, this may involve changing to another virtual machine. In multiprocessors, it is necessary to bind the interrupt to a specific machine and/or address space.

DELETE SECTION 13.6

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 13.6

PROBLEM:

Section 13.6 contains no semantic or syntactic definitions. It consists entirely of exposition and an example. It serves no useful purpose within a standard. As a programming jewel, it should be included in some volume other than the standard definition of the language. Its inclusion serves only to complicate the tasks of compiler implementors and users who have an additional page of material through which to wade.

While there is no objection to the effect of the material, note that an alternative use of a function to implement the conversion could result in a possibly more effective conversion (or possibly a less effective one) depending upon the types being converted and the implementation being used. Using the phrase 'It is necessary to declare a second type, derived from the first' is inaccurate because it fails to mention the alternative use of a user-defined function. Using such a function would not require the use of a second type derived from the first.

IMPORTANCE:

The effect of this proposal, combined with the many other similar proposals dealing with other parts of this standard, is a good indication that a serious effort needs to be undertaken to scrub the standard for clarity, accuracy and consistency.

CURRENT WORKAROUNDS:

Provide good Ada instructors and mentors.

POSSIBLE SOLUTIONS:

The failure to mention the existence of an alternative implementation makes the appearance of this section within a standard misleading, which is an unfortunate adjective to be able to apply to a standard. Since the material adds nothing to the definition of the language, delete the section.

PERMIT SUBTYPES OF TYPE ADDRESS

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 13.7

PROBLEM:

Type ADDRESS is of limited utility. The designers of the language did not allow for convenient pointer operations dealing with dynamically created objects and objects created by declarations. In particular with one possible interpretation of the rules of the language, an application cannot create a record to be inserted into a list using an object declaration.

IMPORTANCE:

The implementability of applications employing pointers can be greatly improved.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Modify the declarations of ADDRESS types to: subtype type_name IS ADDRESS [constraint] where the optional constraints is a subtype indication as in access type declarations. Then at compile time, the implementation can check that assignments to objects of ADDRESS type deal with consistent target types. Allow explicit conversions between ADDRESS subtypes and access types. In particular, with this mod, application developers can instantiate unchecked_conversion functions and receive compile time checking of target consistency. The lack of a constraint to an ADDRESS declaration would provide an untyped pointer capability for those applications who desire such a capability. But the key prospect is providing both additional flexibility and additional type checking for application development.

SORT KEY ATTRIBUTES

DATE: August 11, 1989

NAME: Larry Langdon

ADDRESS: Census Bureau
Room 1377-3
Federal Office Bldg 3
Washington, DC 20233

TELEPHONE: (301) 763-4650
E-mail(temporary): langdonl@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.7.2, Appendix A

PROBLEM:

In every substantial system that we are familiar with (DEC, IBM, SPERRY and BURROUGHS) there is a "system sort" set of utility subroutines. These routines have been developed over a longer time than the Ada compiler itself and are concerned with the highest possible speed and integrity in the sorting process. There is a need for an interface (or at least the means to make one) to such sorting utility subroutines.

The thing preventing us now from creating our own interface to a system sort is the lack of means of gracefully conveying information about the keys of the sort. It is important that the keys be expressible in terms of Ada variable names. Some sort of string expressing key length, type, and starting offset for each key, while certainly possible, would be VERY ugly, involving the user in undesirable, error-prone, and non-portable details of representation.

Note that though this problem is couched in terms of the use of system sort utilities, the problems of accessing indexed-sequential files, merging sorted files, and using or writing generalized searching utilities are very similar. In particular, access to all of those types of utilities
would be made much easier with the possible solution given below.

IMPORTANCE: IMPORTANT

As it is now, any programmer using system sort utilities must be very familiar with the representation of any key fields of any type being sorted. Some of the information needed to define a key field may be obtained from attributes (starting offset and bit length). To do so for a number of keys is very verbose. This detracts from readability and invites hard-to-detect errors (setting up two keys, it would be tempting to copy the statements for the first and make changes...only you might not see all the places to change).

Some of the information needed to define a key field is not obtainable from attributes (sort method). This creates a need for knowledge of the representation for each field used, and for how that maps into the pre-defined methods. This is even more error-prone than the length and offsets, as well as being completely (and non-obviously to a later reader of the code) implementation dependent.

CURRENT WORKAROUNDS:

These are implementation dependent, but they would involve tedious and verbose specification of keys for each sort used. A small saving is possible through encapsulation of key information in a 'KEY' data type, but such KEY objects would still have to be loaded individually. Note that attempts to create a procedure which would give this information for a given key will fail. Even ignoring the lack of any attribute to give sorting method, the "componenthood" of an object (and hence its offset) would be lost at it was passed to the procedure.

POSSIBLE SOLUTIONS:

Possibility 1 (define a standard package spec for sorting interface):

This would be easiest for the user IF the diversity of the sorting utilities on different systems will allow it. It is at least worth pursuing.

Possibility 2 (provide means to allow user-written interfaces):

The impediments to writing an interface to a system sort package would be removed by the addition of the following two things (note that, as mentioned above, these would also facilitate use of indexed-sequential file, merging, and searching utilities):

a) That a new type be defined in the package system called KEY_TYPE. This type might be thought of as a record:

```
record
        stor_unit_offset, bit_offset, bit_length : integer;
        ascending : boolean;
        method : comparison_method;
                -- an implementation-dependent type whose values
                -- enumerate the possible comparison methods:
                -- signed, unsigned, and maybe others.
        end record;
```

where

stor_unit_offset	is the offset of the beginning of the key from the start of the objects of the type being sorted, given in STORAGE_UNITs.
bit_offset	is the offset of the beginning of the key from the start of the storage unit where the key starts, given in bits.
bit_length	is the length of the key, in bits.
ascending	indicates whether the key is ascending or descending.
method	is the comparison method appropriate for this type of object.

b) Two attributes, KEY and DKEY, which returns a KEY_TYPE for any object of a scalar or discrete array type. Each returns the information above for that object (KEY for ascending, DKEY for descending). If the object is a component of a record object (the most common case), the offsets are with respect to the outermost containing record. If the object is not a component of a record object, the stor_unit_offset is zero.

SAMPLE USE

Given KEY and DKEY, a wide variety of sorting utilities (either implemented as interfaces to system sorts or stand-alone) could be written. This would include generic procedures to sort files, arrays in memory, etc. The example given below is a very general one, in that the others could be constructed from it. Conversely, any solution to this problem would have to allow the writing of this (or an equivalent) package.

```
generic
        type obj is private; --
package sort is
        type sort_type is limited private;  -- analogous to file_type
        type keys is array(positive range <>) of key_type;
        procedure open_sort     (s      : sort_type;
                                 k      : keys;
                                 form   : string:="");
                        -- "form" would be implementation-defined
                        -- and could optionally be used for tuning
                        -- a particular sort.
        procedure put_to_sort(s : sort_type;  r : obj);
        procedure get_from_sort(s : sort_type;  r : out obj);
        procedure close_sort(s : sort_type);
        function end_of_sort(s : sort_type)  return  boolean;
        -- etc.
        end sort;
```

The typical pattern of use for this would be:

```
type myrec_type is record
                    fld1 : integer;
                    fld2 : string(1..8);
                    fld3 : boolean;
                  end record;
package this_sort is new sort(myrec_type);     use this_sort;
        mysort : sort_type;
        myrec : myrec_type;
begin           -- the following sorts objects of type myrec_type by their fld2's and, for equal fld2's
                by descending values of the integer fld1.
                open_sort(mysort,keys=>(myrec.fld2'key,myrec.fld1'dkey));
-- etc.
put_to_sort(mysort,myrec); -- as many times as you have records
-- etc.
while not end_of_sort(mysort)
  loop
    get_from_sort(mysort,myrec);
    -- process the record "myrec"
 end loop;
```

DENOTING VALUES OF TYPE ADDRESS IS NOT STANDARDIZED

DATE: August 27, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne, IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 13.7(2)

PROBLEM:

Since type ADDRESS is private, there is no easy, standardized way of representing ADDRESS literals in the language. Examples in the LRM usually assume that values of type ADDRESS are simply represented as integers. Try getting a compiler to accept 16#FFFAAAA# as a valid integer. This value is out of range. This is unacceptable. But each compiler vendor now gets to decide how it will allow the programmer to represent values of type ADDRESS to the compiler.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use negative integers for positive addresses or use the vendor's method (if provided).

POSSIBLE SOLUTIONS:

One solution is to make a new type for representing addresses and declare this in package SYSTEM.

For example:

```
type ADDRESS_SPACE is range 0 .. MEMORY_SIZE;
```

A method for converting from values of type ADDRESS SPACE to type ADDRESS (and vice versa) is also required:

```
function TO_ADDRESS (SOURCE : in ADDRESS_SPACE)
        return ADDRESS;
function TO_ADDRESS_SPACE (SOURCE : in ADDRESS)
        return ADDRESS_SPACE);
```

Another solution is to change the representation of type ADDRESS from private to, say:

```
type ADDRESS is range 0 .. MEMORY_SIZE;
```

This eliminates the need for conversion routines.

Of course, for either solution, it is doubtful that the underlying representation of the numeric values for addresses can support negative numbers. Fortunately, negative address space is very rare in today's computers.

PORTABLE ACCESS TO FLOATING POINT COMPONENTS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: 13.7.3

SUMMARY:

The Ada language defines floating point numbers in terms of a sign, an exponent, and a mantissa, and provides builtin means to establish various properties of these components, but does not provide a means to portably access these components. We propose that Ada incorporate several new functions to allow an Ada programmer to access these components of a floating point number, and also to compose a floating point number with these components.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

Many numeric algorithms on floating point numbers require access to the various components of the floating point number in order to gain efficiency. For example, log (x) is most efficiently computed by separating the exponent and mantissa of x as in the following:

```
log (x)= log (x mantissa*b**x'exponent) =
             log (x'mantissa) + x'exponent*log (b)
```

The square root function sqrt(x) is also most efficiently computed by separating the exponent and the mantissa of x, but for a different reason. The Newton-Raphson iteration converges very quickly for computing the square root, but only if the initial approximation to the square root has the right exponent (± 1); if the exponent is not close, the Newton-Raphson iteration will have to linearly search the exponents to find the correct one before starting to converge on the mantissa. Thus, a bad approximation to the square root could require hundreds of iterations. A few of the right high order bits of the mantissa can also dramatically speed up the square root function.

For these reasons, Ada should provide a portable mechanism for extracting the portions of a floating point number.
There are actually seven different functions required:

Let T be a floating point type.
Let X be an object of type T.
Let radix be the floating point radix, i.e., T'MACHINE_RADIX.
X'MACHINE_EXP yields the exponent of floating point object X as a universal_integer
X'MACHINE_MANTISSA yields the mantissa of floating point object X as a universal_integer
($0 \leq$ X'MACHINE_MANTISSA < radix**(T'MACHINE_MANTISSA))

X'MACHINE_SIGN yields the sign (±1) of object X as a universal_integer
T'NEW_FLOAT (sign,exponent,mantissa) constructs a float of type T from the integer components (T'NEW_FLOAT (-1,0,0) yields +0.0, not -0.0)
X'FLOAT_MACHINE_MANTISSA yields a copy of X with its exponent set to zero.
(1.0/radix ≤ X'FLOAT_MACHINE_MANTISSA ≤ 1.0)
X'FLOAT_MACHINE_EXP yields a copy of X with its mantissa set to 1.0.
X'FLOAT_MACHINE_SIGN yields a copy of X with its exponent and mantissa set to 1.0.
(X= X'FLOAT_MACHINE_SIGN*
X'FLOAT_MACHINE_EXP*
X'FLOAT_MACHINE_MANTISSA)

JUSTIFICATION/EXAMPLES/WORKAROUNDS:

The justification is given above.

One might wish to solve the square root problem given above by using a table lookup for an initial approximation to a square root, where the table is a two-dimensional constant array of floating point numbers, and the exponent is used as the first dimension, and some number of high order bits of the integer mantissa is used as the second dimension. (Of course, in assembly language, one would simply extract some number of high-order bits of the floating point representation, which would presumably include the same information, and use that as the index into a single dimensioned vector of initial approximations.)

The only workarounds are to simulate the above functions using portable or non-portable means. Finding the exponent of a floating point number portably requires a binary search of the exponent space. Once this has been achieved, the other components can be more easily computed, although still not as easily as one might using assembly language.

NON-SUPPORT IMPACT:

Continued inefficiency or non-portability of numeric functions.

POSSIBLE SOLUTIONS:

The component extraction functions discussed above.

DIFFICULTIES TO BE CONSIDERED:

The extraction functions might not be single instruction on some computers, depending upon how closely their model of floating point corresponds to Ada's model.

Some implementations may not be capable of handling a floating point mantissa as a single integer; that implementation may have to define some sort of multiple-precision integer type and return the mantissa in that form.

REFERENCES:

Common Lisp provides an equivalent comprehensive floating point extraction and composition mechanism.

Steele. Guy L., Jr. Common Lisp: the Language. Digital Press, Burlington, MA, 1984, p.218.

FUNCTIONS IMPLEMENTED IN MACHINE CODE

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.,
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 13.8

PROBLEM:

It would be convenient for the user, if code statements were also allowed in function bodies. It should then be considered to permit return statements in such bodies along with code statements.

Casting code inserts that are conceptually functions into a procedure mold is inconvenient and often less efficient.

By providing return statements in a sequence of code inserts, the user could be relieved of the necessity to know the respective calling conventions used by the specific compiler.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Casting the code insert into a procedure encapsulation at the cost of usually non-optimal code (where it matters most).

POSSIBLE SOLUTIONS:

as stated

USABLE MACHINE CODE INSERTIONS

DATE: September 1989

NAME: Randal Leavitt (Canadian AWG #007)

ADDRESS: PRIOR Data Sciences Ltd.
240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: Section 13.8 Machine Code Insertions

PROBLEM:

The Ada standard does not specify adequate features for the machine code insertion capability. This guarantees in effect that every compiler will be different in this area. As a result software reuse and portability are hampered.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Most compilers provide a capability to import modules written in assembler. Using this just about guarantees that another compiler cannot be used. It also encourages low level assembler programming.

POSSIBLE SOLUTIONS:

A discussion of the issues involved and a proposed specification for machine code insertions was given in Ada Letters, Volume VI, Number 6, Pages 54..60.

EXCEPTION HANDLERS IN PROCEDURES CONTAINING CODE STATEMENTS

DATE: October 20, 1989

NAME: Thomas J. Quiggle

ADDRESS: Telesoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (619) 457-2700 ex. 158

ANSI/MIL-STD-1815A REFERENCE: 13.8

PROBLEM:

A procedure containing Machine Code Insertions (hereafter referred to as an MCI procedure) may affect the state of the machine such that the surrounding Ada code could not reestablish correct state following an exceptional exit from the MCI procedure. Examples of such state changes include:

* The MCI procedure disables interrupts and begins to communicate with a memory-mapped device. The memory mapped device does not respond, violating bus protocols. The resulting processor exception is mapped to an Ada exception that propagates from the point of the offending code statement. The surrounding code then executes with interrupts (unintentionally) disabled.

* An MCI procedure is created to permit calling routines written in a language not recognized by the Ada compiler. The target language has register usage conventions that are incompatible with the calling conventions of the Ada compiler (for example, the Ada compiler reserves a register for use in in referencing up-level data, the target language uses the same register for some completely different purpose). The target language interface does not trap certain processor exceptions that map to the Ada Numeric_Error exception. The MCI subprogram serves exclusively to maintain the correct register usage conventions across calls to the target language. If an exceptional condition occurs in the destination subprogram, the proper register state is restored by the MCI procedure prior to returning to the surrounding Ada code.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

No workaround exists for the case that the MCI subprogram modifies the register state in a manner that prevents exception propagation beyond the MCI subprogram. For other cases, MCI procedure must be nested in an enclosing Ada subprogram that contains a handler. For example:

```
procedure CALLS_MCI_PROCEDURE is

    PRE_CALL_STATE : APPROPRIATE_TYPE;

    procedure RECOVER_STATE is
```

```
        begin
                ... -- code statements to restore proper state
                -- (assumes MCI implementation that facilitates
                --  access to up-level Ada objects)
        end RECOVER_STATE;

        procedure MCI_PROCEDURE is
        begin
                ... code statements to save proper state in PRE_CALL_STATE.
                ... code statements that may result in an ada exception
        end MCI_PROCEDURE;

begin
        MCI_PROCEDURE;
exception
        when others => RECOVER_STATE;
end CALLS_MCI_PROCEDURE;
```

The above series of nested subprograms is unnecessarily cumbersome, and therefore error-prone.

POSSIBLE SOLUTIONS:

In section 13.8, paragraph 3, eliminate the requirement that for a procedure body containing code statements "no exception handler is allowed."

An implementation should be permitted to place appropriate restrictions on the allowable forms of exception handlers in MCI procedures. Specifically, and implementation should be permitted to allow only an others handler, only handlers for predefined exceptions, optional reraise statements; or to disallow exception handlers altogether. This ensures that an implementation can preserve the optimal subprogram calling and exception handling conventions for a given architecture (the implementation of this capability should not adversely affect overall code quality).

It is possible that an implementation of Ada exception handling would require calling conventions that are incompatible with those for MCI procedures (i.e. a dedicated register that "points to" the current handler, which if modified, prevents meaningful exception propagation). Consequently, exception handlers should not be required functionality. Implementations which can support exception handlers in MCI procedures should not be prohibited from providing such support.

CODE STATEMENTS IN FUNCTION BODIES

DATE: October 20, 1989

NAME: Thomas J. Quiggle

ADDRESS: Telesoft
5959 Cornerstone Court West
San Diego, CA 92121

TELEPHONE: (619) 457-2700 ex. 158

ANSI/MIL-STD-1815A REFERENCE: 13.8 paragraph 3, 6.5 paragraph 2

PROBLEM:

The language standard does not permit code insertions in the body of an Ada function. This requires unnatural, and often inefficient, programming practices for use of Machine Code Insertions (MCIs).

It is already impossible to guarantee that Ada semantic rules will not be violated by code statements. For example, a code statement could read an out parameter, write to an in parameter or constant, assign to an object of a limited type, violate type constraints, etc. To disallow code statements in functions simply because a function is semantically required to contain a return statement is overly restrictive.

Programmers writing code statements in procedures need explicit knowledge of the compiler's parameter passing and register usage protocols. Requiring understanding of function return protocols does not significantly increase the complexity of using code statements, and provides significant additional capability.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Natural applications of MCI functions are forced to define Ada functions with nested MCI procedures. For example, given some (non-integer) type Unsigned_Integer, an addition function for the type can be defined as:

```
function "+"(LEFT, RIGHT : UNSIGNED_INTEGER)
                    return      UNSIGNED_INTEGER is

    RESULT : UNSIGNED_INTEGER;

    procedure ADD(LEFT, RIGHT : in  UNSIGNED_INTEGER
                                SUM       : out UNSIGNED_INTEGER) is
    begin
        ... -- code statement(s) to compute LEFT + RIGHT
        ... -- and store in SUM
    end ADD;

begin
    ADD(LEFT, RIGHT, RESULT);
```

```
        return RESULT;
end "+";
```

The efficiency of these nested subprograms is dependent on a high level of optimization.

POSSIBLE SOLUTIONS:

Modify the first sentence of section 13.8, paragraph 3, to read:

> A code statement is only allowed in the sequence of statements of a subprogram body.

thus allowing code statements in functions.

Additionally, section 6.5, should be modified to permit MCI functions with no return statements, and to allow leaving an such MCI function without raising PROGRAM_ERROR.

UNCHECKED TYPE CONVERSIONS

DATE: September 18, 1989

NAME: Linda Burgermeister

ADDRESS: ITT Avionics
Dept. 73813
390 Washington Ave.
Nutley, N.J. 07110

TELEPHONE: (201) 284-3920

ANSI/MIL-STD-1815A REFERENCE: 13.10.2

PROBLEM:

Ada does not place any guidelines on the result of an unchecked conversion. Even though an application has explicitly defined the layout of a type with a representation specification, some compilers currently generate hidden fields and unique storage representations that are unknown to the application.
This does not affect normal operations because the compiler makes the appropriate data storage adjustments. However, when an unchecked conversion is performed on such objects, the result may contain unexpected erroneous data because of these hidden fields and unique storage methods.

Some compilers perform the unchecked conversion with the expected results, and the "hidden" fields and unique storage methods remain invisible to the user. Other compilers do not perform the expected conversions. For example, one compiler stores variant records in two parts with a pointer in the invariant part pointing the non-contiguous variant part. An unchecked conversion on this record results in the retention of the pointer and whatever happens to be in the contiguous memory locations, not the desired variant record.

The Ada LRM should specify that unchecked conversions between types defined by representation specifications should be converted between the user defined representations (without any unique compiler defined or random contiguous data).

IMPORTANCE: IMPORTANT

The lack of Ada unchecked conversion guidelines forces the user to have knowledge of how the compiler lays out the fields of a type, even though the application may have imposed a specific bit representation.

CURRENT WORKAROUNDS:

If a compiler does not return the expected value as a result of an unchecked conversion, a user must create special procedures that perform the "unchecked conversion" of the objects. These routines would have to compensate for the compiler generated fields and storage methods.

POSSIBLE SOLUTIONS:

The results of an unchecked conversion should contain only the data defined by the source code type representation specifications. If a compiler uses additional hidden fields or unique storage methods, then the results should be modified appropriately so that they remain invisible to the application.

ADA COMPILERS CAN ALLOW THE GENERIC FUNCTION UNCHECKED_CONVERSION TO BE INSTANTIATED WITH THE TYPE_MARK OF A PRIVATE TYPE PROVIDED AS A GENERIC_ACTUAL_PARAMETER

DATE: September 28, 1989

NAME: Kenneth E. Rowe and David A. Silberberg

ADDRESS: Attention C333
9800 Savage Road
Fort George G. Meade, MD 20755-6000

TELEPHONE: (301) 859-4494

ANSI/MIL-STD-1815A REFERENCE: 13.10.2(2,3), 7.4(1), 7.4.4(9), 12.3

PROBLEM:

Ada compilers can allow the generic function UNCHECKED_CONVERSION to be instantiated with the type_mark of a private type provided as a generic_actual_parameter. This allows users to subvert the intended benefits of using private and limited private types. Additionally, this causes an contradiction in the definition of private types since 7.4(1) states "The declaration of a type as a private type in the visible part of a package serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself)."

IMPORTANCE: ESSENTIAL

If this problem is not resolved, it can lead to vulnerabilities in Ada-based mission critical and safety critical systems. Additionally, it calls into question the soundness of the abstraction/information hiding capabilities of the Ada language.

CURRENT WORKAROUNDS:

No cost effective methods are currently available. The only solution outside of the Ada language is to provide assurance that unchecked_conversion cannot be instantiated on private types. This would require a) a language pre-processor to detect and prevent usage coupled with access controls to assure that the pre-processor cannot be circumvented or b) disqualification of compilers from critical domains if they allow the private type instantiation.

POSSIBLE SOLUTIONS:

To modify 13.10.2(2) to not allow the use of private types as generic_actual_parameters for unchecked_conversion. This change should also be reflected in 12.3 (Generic Instantiation).

CONSTRAINT CHECKING AFTER UNCHECKED CONVERSION AND IO

DATE: October 27, 1989

NAME: Mike Glasgow

ADDRESS: IBM Systems Integration Division
9231 Corporate Blvd.
Rockville, MD 20850

TELEPHONE: (301) 640-2834

ANSI/MIL-STD-1815A REFERENCE: 13.10.2, 14.2.2;4, 14.2.4;4, 14.3.5;10

PROBLEM:

A common model for communication between separate Ada programs in a distributed application is to have a service that converts a typed message (usually a record type) to an untyped string of bytes when sending a message, and vice versa when receiving a message. Typically there is some unique identifier passed with the string of bytes to identify the appropriate message type. The process necessarily involves unchecked conversions to prevent the communication service from having to have knowledge of every message type.

The problem is that when the string of bytes is converted back to a message type, there is no convenient way to verify that the resultant object adheres to the constraints of the corresponding type. The unchecked conversion, as advertised, is unchecked. Assignment of the resultant object to another object of the same typically does not cause checks to be emitted, nor does assignment of a component of the object to another object.

The same situation occurs with pre-defined IO. The GET operations for text IO as well as the READ operations for sequential and direct IO do not require the DATA_ERROR exception to be raised if the element read cannot be interpreted as a value of the target type (the exception is optional, and many vendors simply perform no checks other than possibly a length check).

It is crucial in both situations to be able to verify the data and prevent corrupted data from entering the application.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The only workaround that will always work is to manually code the constraint checks for each component of the message, e.g., is this integer_type component >= integer_type first and <= integer_type'last. Note that this type of checking is identical to what a compiler will generate when circumstances require the checks.

POSSIBLE SOLUTIONS:

Two possibilities:

1. Half_checked_conversion/Enforced DATA_ERROR; Providing a function similar to unchecked conversion that does constraint checking for the target type and requiring DATA_ERROR to be supported would solve the problem. (Credit for the half_checked conversion idea goes to David Brookman).

2. Pragma Check: If after an unchecked_conversion or an IO read operation one could code:

```
a := a;

  or

a.all := a.all;
```

and be guaranteed that constraint checks would be generated, this would also provide a solution. Since in most cases these statements would not result in the checks being generated, a new pragma could be supported that would require/insure that they are:

```
pragma check;
a := a;
```

Both of these solutions can be implemented such that previously existing code is upwards-compatible with Ada9X.

INTERFACE TO OTHER ANSI LANGUAGES

DATE: September 26, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@TREES.DNET.GE.COM

ANSI/MIL-STD-1815A REFERENCE: 13.9

PROBLEM:

In constructing an application, it may become necessary to merge code written in Ada with code written in other languages. However, since the description of the language interface is implementation-dependent, code of this type is non-portable by definition.

This applies even though the other code may also be defined by ANSI or military standards. It would therefore be desirable to have a standardized interface specification for other languages defined by ANSI standards.

C and FORTRAN are probably the two languages for which a specification would be most desirable. Other languages, such as Pascal and COBOL, would also be desired, if practical.

It is understood that some language attributes may not be translatable into Ada and therefore could not be supported by this type of interface. It is also understood that such features would not be mandatory, but that their existence would be implementation dependent.

IMPORTANCE: IMPORTANT

I feel that such an addition would be important because it would increase the transportability of applications using Ada code. By standardizing the interface, systems composed of code in the C and Ada programming languages could be transferred without modification.

CURRENT WORKAROUNDS:

Interfaces to programs in other programming languages are defined by a implementation-dependent pragmas. These facilities can be used to create code with modules written in different programming languages.

POSSIBLE SOLUTIONS:

My view is that the pragmas describing the nature of the interface should be clearly understandable to the programmers in Ada and in the foreign language. I believe that this can be done by creating the following types of pragmas. (This section is not intended to serve as a formal language description, but simply as an expansion on my initial thoughts.)

First pragma:

pragma FOREIGN_TYPE (language_name, datatype_name, datatype_description)

language_name is the name of the foreign language to which the interface is desired

datatype_name is the name of a data structure in language language_name written according to the rules of language language_name.

data types would be able to be used in later pragmas of this type and would also be usable in other packages to which the code containing the pragma is visible. When used in other packages, the name of the source of the statement would be indicated using dot notation in the same way as any other type that is passed between packages.

For a given language, there would be a set of predefined data types. Where the data type is defined by more than a single word, underscores would be placed between the words in the name of the data type name. In this way, predefined data types for the C programming language would include INTEGER, SHORT_INTEGER, FLOAT, and LONG_FLOAT. Predefined types for FORTRAN would include INTEGER, REAL, and DOUBLE_PRECISION.

Since there would be predetermined types for each foreign language, this type of pragma would be optional when dealing with simple types.

This pragma would also be able to construct complex types such as records and arrays. There shouldn't be much trouble handling one-dimensional arrays and records composed of simple data types. More complex structures would have to be examined for their suitability for the automatic translation that would have to be carried out when going between programming languages.

The type declaration formats in Ada for records and arrays are simple enough that they should be readable to programmers whose background is in other languages. For records, the description would start with the word record and end with the words "end record". For arrays, the description would begin with the word "array" and end with the type name of the component. The rest of the syntax would then be taken from the Ada syntax for the type declaration.

If the translation process would require placing a string in a structure that is too short, the value would be truncated. Attempts to put data fields that are too long would result in the field being padded with nulls.

Second pragma:

pragma FOREIGN_RETURN (subprogram_name, datatype_name)

subprogram_name is the name of the subprogram for which the interface is being defined
datatype_name is the name of the data structure defined by the FOREIGN_TYPE and is a description of the data structure returned by the procedure.

Third pragma:

pragma FOREIGN_CALL (subprogram_name, (list of datatype_name), (list of calling methods))

The list of datatype_name's would be a list of the datatype_name that define the structures

in the foreign language.

The list of calling methods would be a set of values separated by commas and surrounded by parenthesis. Two of the allowable values in this list would be VALUE and REFERENCE. VALUE would indicate that the value is placed in the argument list while REFERENCE would indicate call by reference passing the address of the value. The defaults would depend on the language being used for the interface, and would be defined for the languages for which the interface was defined.

Since defaults would exist for the calling methods, the list of calling methods is an optional entry. This pragma could also be combined with the existing pragma INTERFACE by adding optional arguments to the syntax.

For additional references to Section 13. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

NUMBER	TITLE	SECTION
0024	SEPARATION OF EXPONENT AND MANTISSA	4-4
0037	CONTROL OF CLOCK SPEED AND TASK DISPATCH RATE	9-28
0107	CONFIGURING CALENDAR.CLOCK IMPLEMENTATION	9-30
0109	DISTRIBUTED SYSTEMS	9-12
0220	ENUMERATION LITERAL INTEGER CODES	3-118
0263	WHEN A CONSTRAINT ERROR IS TO BE RAISED	4-45
0283	MACHINE CODE INSERTIONS	10-37
0286	INTERRUPTS	11-18
0316	IMPROVED INTERRUPT HANDLING	6-28
0357	DECIMAL	3-189
0365	IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (I)	3-121
0369	ADA SUPPORT FOR ANSI/IEEE STD 754	3-103
0377	REQUIREMENT TO ALLOW PARTIONING OF ADA PROGRAMS OVER MULTIPLE PROCESSORS IN DISTRIBUTED/PARALLEL/MULTI-PROCESSOR ENVIRONMENT	8-40
0393	VISIBILITY OF ARITHMETIC OPERATIONS	8-46
0432	IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (II)	3-124
0478	PROTECTION AGAINST VIRUSES AND TROJAN HORSES	9-65
0480	RENDEZVOUS BETWEEN INDEPENDENT PROGRAMS	9-67
0640	ACCESSING CHUNKS OF BIT-VECTORS AND BIT-ARRAYS	4-35

ADA 9X REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A

SECTION 14. INPUT-OUTPUT

OUTPUT OF REAL NUMBERS WITH BASES

DATE: May 23, 1989

NAME: Jurgen F H Winkler

ADDRESS: Siemens AG ZFE F2 SOF3
Otto-Hahn-Ring 6
D-8000 Munchen 83
Fed Rep of Germany

TELEPHONE: +49 89 636 2173

ANSI/MIL-STD-1815A REFERENCE: 14.3.8

PROBLEM:

Allow output of real numbers with bases from 2 to 16

Ada allows the notation of real numbers with bases from 2 to 16 in the source program and in the input for the GET procedures, but does not allow the specification of a base in the PUT procedures.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Write new PUT procedures.

POSSIBLE SOLUTIONS:

```
subtype NUMBER_BASE is INTEGER range 2..16;

DEFAULT_BASE : NUMBER_BASE: = 10;

procedure PUT ( FILE     :in FILE_TYPE;
                ITEM     :in NUM;
                FORE     :in FIELD: = DEFAULT_FORE;
                AFT      :in FIELD: = DEFAULT_AFT;
                EXP      :in FIELD: = DEFAULT_EXP;
                BASE     :in NUMBER_BASE: = DEFAULT_BASE);
```

I/O SYSTEM SETUP

DATE: August 12, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 14

PROBLEM:

The I/O system is set up as a collection of procedures rather than a collection of task entry points.

IMPORTANCE:

CONSEQUENCES:

A task which is waiting for keyboard input will have its flow of control seized by Text_IO.Get; since task starvation is permitted (LRM 9.8.5), this can have the effect of preventing useful work from being accomplished by other tasks while a user is getting around to providing the next keystroke.

CURRENT WORKAROUNDS:

If one is willing to venture into machine dependencies, it is possible to set up a task which will respond to an interrupt from the keyboard, but task suspension pending an I/O interrupt should be writable in a portable manner.

POSSIBLE SOLUTIONS:

Set up entry calls for I/O; this would permit a task to suspend waiting for a rendezvous with Text_IO.Get, and thereby allow other tasks to run instead. Another approach might be to require fair task scheduling in such a way that tasks which are awaiting I/O will give up the CPU to tasks which are runnable.

NO-WAIT I/O

DATE: June 21, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: 14

PROBLEM:

Many times, real-time systems need to issue an I/O request, do some other work and then check the status of the request. In other words, an I/O request must be processed asynchronously to the usual line of processing. No such facility exists in the language.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Establish an I/O server task which tracks and issues I/O requests. This is quite cumbersome.

POSSIBLE SOLUTIONS:

Include an additional boolean (defaulted) parameter to all Text_IO, Direct_IO, and Sequential_IO procedures (not functions) which will indicate the form of I/O required - synchronous or asynchronous.

INTERACTIVE TERMINAL INPUT-OUTPUT

DATE: March 1, 1989

NAME: Mike Curtis (and members of the Ada-Europe Environments Working Group; endorsed by Ada UK and Ada-Europe)

ADDRESS: ICL
Eskdale Road
Winnersh
Wokingham
Berkshire
RG11 5TT
United Kingdom

TELEPHONE: +44 734 693131

ANSI/MIL-STD 1815A REFERENCE: 14

PROBLEM:

The facilities for Input-Output provided by the TEXT_IO package are inadequate for handling modern terminals and programming the sort of user-interfaces expected in modern software packages.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

One possibility is that the low-level functions of the terminal, for example escape codes for screen control and values returned by function keys, can be defined, and then TEXT_IO can be used. This can give some limited control but it becomes extremely complex if terminal independence is required and it gives no help with use of the graphics characters for line drawing.

The other possibility is to use a different language.

POSSIBLE SOLUTIONS:

A new package "TERMINAL_IO" can be defined, with the same status as TEXT_IO in that it can be omitted if it is not relevant for a particular compiler/machine combination. It should include basic screen operations, such as positioning the cursor and clearing and scrolling all or part of the screen, recognition of function keys and possibly higher level operations such as the creation and manipulation of text window. Provision should be made for the use of the full extended character set.

INPUT-OUTPUT

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: 14

PROBLEM:

Input-output methods are architecture dependent for embedded computers. Many of the features supported in Ch 14 are primarily directed toward what is needed for validation on mainframes/ mini computers. As the requirements are stated, the embedded systems developer has to obtain Ch 14 provisions even though they are useless for the intended applications. The residue of those runtimes for IO may even be contained in the applications when they are not used. The "main frame" mind set is not useful for embedded systems with special IO needs. Because of this, many applications opt for another implementation language rather than use Ada.

IMPORTANCE:

High for embedded systems and for information systems

CURRENT WORKAROUNDS:

Create your own IO functions and never reference or import the Ada provided IO package.

POSSIBLE SOLUTIONS:

We need quick response times for IO for data that is sampled at rates far greater than 50 Hz. We do not have files to be opened, closed, or created. We also may not have text--if we do, it is not the kind that interfaces with standard IO devices. The IO mechanism needs to support atomic transactions and/or double buffering. The interfaces to the "standard" packages are far too inefficient for the applications and have no meaning. Currently, much hardware supports memory mapped IO and double buffering. The language standard should allow a better approach without have to revert to assembler language and mounds of code--a particularly difficult maintenance problem. Therefore, we recommend the following:

1. Make chapter 14 optional

2. IO was added to the language during the standardization process and did not have an adequate review from industry. In particular, it is not well integrated into the language. The packages are inefficient. Most Ada'83 solutions show a tendency to "write lots of Ada code to show off" which does not fix the problem of integrating IO into the language. The current method leads to great amounts of overhead in applications. Simple BNF should be created with at most two key words

for input and for output. There's far too much code in Ch 14 for an LRM definition--this greatly hampers its suitability for a wide variety of applications.

3. Fix the problem and do not add to the difficulties by fixing only the various symptoms.

4. Delete IO as it is in many languages. It will not be portable among the architectures anyway. The compiler vendors are ill-equipped to provide standard packages for the embedded IO applications. The exception handling is also device and applications specific and the solutions in the LRM are not suitable for embedded processing.

5. Delete low-level IO

FILE SYSTEM FUNCTIONS

DATE: March 21, 1989

NAME: Larry Langdon

ADDRESS: Census Bureau
Room 1377-3
Federal Office Bldg 3
Washington, DC 20233

TELEPHONE: 301-763-4650
E-mail(temporary): langdonl@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: 14.1, paragraphs 1, 2, and 10.

PROBLEM:

There are some types of information about files and the file system which transcend the individual input-output packages. They are best expressed in terms of external files. Functions to return these types of information should not simply be replicated in the individual packages. You may not know the type of a file for which you are seeking information and hence, you may not know what package to use.

Another way of expressing this language issue is to say that there is a frequent need for directory information about a file, and that this information should be obtainable without opening the file for I/O. Of course, the directory information is machine dependent. The choice of functions is subject to reasonable debate, but there is a "core" which should be present on any machine with a file system.

Examples from actual experience:

a) You are implementing a "copy" style operation to replicate an arbitrary file. The actual data transfer can be done in some low-level, block-by-block fashion and is not a problem. Without being able to replicate the directory information for the source file, the created file may well be unusable. What is needed is a FORM function for an external file (i.e., independent of a particular I/O package since the source file is of unknown type). This information could come from a function such as:

```
function FILE_FORM  (name : string) return string;
```

b) You need to know whether a file exists. Its type is irrelevant and unknown. This information could come from a function such as:

```
function FILE_EXISTS  (name : string) return boolean;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: low-level, system-dependent programming.

POSSIBLE SOLUTIONS:

This information could be provided as a "FILE_SYSTEM" package of functions. The functions provided should include (but not necessarily be limited to) the following:

```
function FULL_FILE_NAME   (name : string) return string;
function FILE_EXISTS      (name : string) return boolean;
function FILE_FORM        (name : string) return string;
function CAN_READ         (name : string) return boolean;
function CAN_WRITE        (name : string) return boolean;
function CAN_EXECUTE      (name : string) return boolean;
```

Suggested semantics for each of the functions above are:

FULL_FILE_NAME
: If the string parameter refers to the name of an existing file, then return its full name; else mimic a CREATE operation on that name and return the full name of the file that would result if the CREATE were actually done (but don't do it). If a CREATE (with the mode and form defaulted) would raise a NAME_ERROR exception, then this function should raise the same exception.

FILE_EXISTS
: This is covered in Language Issue 11, but note that it makes more sense to place FILE_EXISTS in this collection or package since its result does not depend on what I/O package it might be contained in.

FILE_FORM
: If the file named by the string parameter does not exist, this function would raise an exception. Otherwise, it would return the same result as the FORM function would return if the file named were opened with an appropriate I/O package (sequential, direct, or text_io). The definition of "form" in Chapter 14 of the LRM seems to assure that this is well-defined (i.e., independent of I/O package). The intent is that FILE_FORM("XYZ") contain all the information necessary to create an exact (albeit empty) copy of "XYZ".

CAN_READ
: This would return true if and only if the file named by the string parameter exists and could be opened for reading (i.e., no protection, file locking, etc. prohibitions exist) at the point of the function call. An exception would result if the file were already open.

CAN_WRITE
: Analogous to CAN_READ

CAN_EXECUTE
: Analogous to CAN_READ

Many other functions deserve consideration for inclusion in this "FILE_SYSTEM" package. A couple of examples are functions to return file size and creation date.

NOTE: This proposal (in slightly different, but substantively identical form) was approved as an Ada Language Issue (LI53) by the Ada Language Issues Working Group of SIGAda. The final vote, taken March 1, 1989, was: 11 in favor; 0against.

ASSIGNMENT FOR TEXT_IO.FILE_TYPE

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 14.1

PROBLEM:

Assignment for TEXT_IO.FILE_TYPE

In some situations one needs support for assignment on FILE_TYPE. For instance to build a stack of default files, such that for example after temporary redirection to another device, interactive text I/O resumes from the point before the redirection was made.

Rationale by Alsys S.A., 1986, page 371:

> A more general requirement might be to surround the local use of a default file by statements which preserve and then restore the existing default file.

Limited private types for TEXT_IO.FILE_TYPE does not allow stack-wise treatment of STANDARD_OUTPUT and STANDARD_INPUT. Deceivingly, the rationale refers to the functions CURRENT_INPUT to indicate that the programmer could solve this problem. We do not see how this could be implemented because of the absence of assignment.

IMPORTANCE:

CURRENT WORKAROUNDS:

Use a type access to FILE_TYPE, and hope that anyone who needs this assignment already used this type. Implies project wide coordination.

POSSIBLE SOLUTIONS:

This problem could be solved by adding PUSH_STANDARD_INPUT and POP_STANDARD_INPUT operations, but of course, other examples exist. For instance, when one needs to support a lexical stream that reads tokens from a FILE_TYPE in a context where the number of open lexical streams is statically unknown (such that generics cannot be used), one needs to connect the lexical stream data type to a FILE_TYPE value. This can only be done if everyone using the package uses a type T_NEW_FILE_TYPE is access TEXT_IO.FILE_TYPE, implying allocation of a FILE_TYPE object before opening the file. But of course, this is impossible for STANDARD_INPUT, and hence this stream needs to be considered a

special case, known to the analyser.

Conclusion: support of FILE_TYPE assignment would help designing clean software. Assignment could be implemented by using an access type to the actual buffer, and a reference count to decide when a CLOSE operation should really close the stream.

MANDATED DISK I/O SUPPORT FOR VARIANT RECORD TYPES WITH THE DIRECT_IO AND SEQUENTIAL_IO PACKAGES

DATE: October 11, 1989

NAME: Jerry W. Rice

ADDRESS: 660 Arboleda Drive
Los Altos CA 94022

TELEPHONE: Work: (415) 969-1898,
Home: (415) 969-2302

ANSI/MIL-STD-1815A REFERENCE: 14.1, 14.2, 14.2.2, 14.2.4, Appendix F

PROBLEM:

Ada does not mandate that the DIRECT_IO and SEQUENTIAL_IO packages provide read and write support for variant record objects (unconstrained record type with default discriminant assignments). The current Ada language definition (and validation process)allow implementations to raise the "USE_ERROR" exception at runtime when the DIRECT_IO and SEQUENTIAL_IO packages are instantiated with a variant record type, and the Open, Read, or Write procedures are invoked. Some implementations, while supporting this capability, require the presence of an implementation defined FORM string when creating a variant record cause significant portability problems in software that requires variant record io, and that also must run on different targets (ie, different compiler/runtime implementations).

IMPORTANCE: ADMINISTRATIVE (or possibly IMPORTANT?)

CURRENT WORKAROUNDS:

There are at least four obvious workarounds, none of which are desirable. The first is to either utilize routines written in another language or to call implementation specific functions. The second is to convert all data to ascii on output, and back to binary on input. The third is to kludge an input-output package body that uses UNCHECKED_CONVERSION heavily. And the fourth is to look for another compiler vendor whose product supports variant record io for your target.

POSSIBLE SOLUTIONS:

I recommend that all implementations of the DIRECT_IO and SEQUENTIAL_IO packages (that provide support for disk i/o) be required to support variant io, and that the FORM string not be required (else, a standard FORM string format should be defined relating to this feature that all implementations must conform to). This will provide a greater degree of portability for systems written in Ada. I encountered this compiler limitation on a large DOD Ada project where the source code was required to compile and run on three different target systems, and it has been a significant problem area. Now, for some specific solution suggestions. For memory-resident variant record objects, most (if not all) compilers/runtimes will automatically allocate the memory required for the largest variant case. A compiler/runtime should be able to do the same for the file elements stored within the external file as well. While this may cause less than optimal diskspace utilization, the designer/programmer can many times mitigate this problem by carefully

defining the variant record type being used. To reiterate, I suggest that you more strongly regulate the semantics of the DIRECT_IO and SEQUENTIAL_IO generic packages. A possible wording for a revised Chapter 14 definition of these packages might be on the order of,

"...The type argument supplied when instantiating DIRECT_IO and SEQUENTIAL_IO must be a type meeting the following conditions. It must be a type

(1) That is not limited, and
(2) Whose full declaration is neither an access type nor contains subcomponents of an access type, and
(3) Is either a constrained (sub)type, or a variant record type (having default initial discriminant values).

..."

The language lawyers can certainly come up with more formal and precise wording, but the basic idea should be clear. Ne last thought: it might be desirable (or necessary) to define a new attribute operator. It would be very similar to T'CONSTRAINED, but it would return TRUE when the type being tested were a variant record type, and FALSE otherwise. It might be called T'VARIANT. This would be useful within a generic package body to determine if an actual type supplied during package instantiation for a formal private type parameter is actually a variant record type. If requested, I would be happy to provide further analysis and suggestions concerning this matter.

NEED FOR SUPPORT FOR ASSIGNMENT ON FILE_TYPE

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belguim

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 14.1

PROBLEM:

In some situations one needs support for assignment on FILE_TYPE. For instance to build a stack of default files, such that for example after temporary redirection to another device, interactive text I/O resumes from the point before the redirection was made.

Rationale by Alsys S.A., 1989, page 371:

> A more general requirement might be to surround the local use of a default file by statements which preserve and then restore the existing default file

Limited private types for TEXT_IO.FILE_TYPE does not allow stack-wise treatment of STANDARD_OUTPUT and STANDARD_INPUT. Deceivingly, the rationale refers to the functions CURRENT_INPUT to indicate that the programmer could solve this problem. We do not see how this cold be implemented because of the absence of assignment.

IMPORTANCE:

CURRENT WORKAROUNDS:

Use a type access to FILE_TYPE, and hope that anyone who needs this assignment already used this type. Implies project wide coordination.

POSSIBLE SOLUTIONS:

This problem could be solved by adding PUSH_STANDARD_INPUT and POP_STANDARD_INPUT operations, but of course, other examples exist. For instance, when one needs to support a lexical stream that reads tokens from FILE_TYPE in a context where the number of open lexical streams is statically unknown (such that generics cannot be used), one needs to connect the lexical stream data type to a FILE_TYPE value. This can only be done if everyone using the package uses a type T_NEW_FILE_TYPE is access TEXT_IO.FILE_TYPE, implying allocation of a FILE_TYPE object before opening the file. But of course, this is impossible for STANDARD_INPUT, and hence this stream needs to be considered a special case, known to the analyzer.

CONCLUSION:

Support of FILE_TYPE assignment would help designing clean software. Assignment could be implemented by using an access type to the actual file buffer, and a reference count to decide when a CLOSE operation should really close the stream.

INTEROPERABLE IO

DATE: October 21, 1989

NAME: Allyn M. Shell

ADDRESS: AdaCraft, Inc.
4005 College Heights Dr.
University Park, MD 20782

TELEPHONE: (301) 779-6024

ANSI/MIL-STD-1815A REFERENCE: 14.2

PROBLEM:

The input/output features of Ada for the instantiated generic IO packages (Sequential_IO and Direct_IO) are woefully non interoperable. Data stored on tape by one program cannot be read by other programs unless the compilers that produced the two programs were both Ada compilers of the same vendor, version and hardware platform. Similar problems exist in telecommunications between programs on different machines. It is even worse if the data was produced by or is to be used by a program not written in Ada.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Use Text_IO. (Even that does not always work around the problem.)

POSSIBLE SOLUTIONS:

Either define a specific IO packages for interoperable sequential IO and interoperable direct IO or make the requirements for the form of the data type more strict when it is used in Sequential_IO and Direct_IO. Minimally, do not allow the "dope vectors" (which precede the data areas in array and record object storage areas) to be included when arrays and records are used in the instantiation of these generic packages.

PROCEDURE TO FIND IF A FILE EXISTS

DATE: October 13, 1989

NAME: David A. Smith, on behalf of SIGAda Ada Language Issues Working Group

ADDRESS: Hughes Aircraft Company, A1715
16800 E. Centre Tech Parkway
Aurora, CO 80011

TELEPHONE: (303) 344-6175
E-mail: dasmith @ ajpo.sei.cmu.edu or
E-mail: smith @ cel860.hac.com

ANSI/MIL-STD-1815A REFERENCE: 14.2.1

PROBLEM:

The following comment represents discussions that took place at the Nov 1986 and Jan 1987 meetings of the Ada Language Issues Working Group (ALIWG). It was agreed that a subprogram is needed in the three predefined I/O packages that, given a string, would indicate whether or not a file by that name exists.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The work-around now is to attempt to open the file, and then if that succeeds, immediately close it. If the file does not exist, an exception is raised:

```
with TEXT_IO;

function FILE_EXISTS (FILENAME : in STRING) return BOOLEAN is FILE :
TEXT_IO.FILE_TYPE;
begin
    TEXT_IO.OPEN (FILE, TEXT_IO.IN_FILE, FILENAME);
    TEXT_IO.CLOSE (FILE);  -- only reached after successful open        return TRUE;
exception
        when others => return FALSE;
end FILE_EXISTS;
```

There are several flaws in this "home-grown" solution. First is that any program using FILE_EXISTS must "with" it separately from TEXT_IO. Furthermore, the above example only works with TEXT_IO files. Since SEQUENTIAL_IO and DIRECT_IO are generic, a generic FILE_EXISTS must be made available and instantiated in parallel. This is clumsy at best.

POSSIBLE SOLUTIONS:

The approximate form of the function would be:

```
function FILE_EXISTS (FILENAME : in STRING) return BOOLEAN;
```

This upward-compatible function would provide a capability that is needed but currently lacking. There was concern as to the exact semantics to be implemented. Semantics requiring an "open then close" implementation are not desired, since under UNIX and perhaps other systems, this would update the time stamp on the file. FILE_EXISTS should be side-effect free.

An alternative semantics was suggested: "return true if an attempt to open would succeed". In this case, it was suggested that a more descriptive name would be appropriate. A true response to a "file exists" query may not be sufficient information, if the file cannot be opened for some other reason.
The following name was suggested, although there was no agreement on the exact name:

```
function CAN_FILE_BE_OPENED (MODE : in FILE_MODE;
                             NAME : in string;
                             FORM : in string) return BOOLEAN;
```

To fully test if the file can be opened, this function requires the "form" and "mode" parameters.

FILE "APPEND" CAPABILITY

DATE: October 13, 1989

NAME: David A. Smith, on behalf of SIGAda Ada Language Issues Working Group

ADDRESS: Hughes Aircraft Company, A1715
16800 E. Centre Tech Parkway
Aurora, CO 80011

TELEPHONE: (303) 344-6175
E-mail: dasmith @ ajpo.sei.cmu.edu or
E-mail: smith @ cel860.hac.com

ANSI/MIL-STD-1815A REFERENCE: 14.2.1

PROBLEM:

Ada has no portable, implementation-independent method for opening an output file in "append" mode. This is a feature that is frequently required in applications and is efficiently supported on many operating systems. Since the raison d'etre of Chapter 14 is to define standard interfaces to important features like this, it is proposed to include this feature.

The following comment represents discussions that took place at the Nov 1986 and Jan 1987 meetings of the Ada Language Issues Working Group (ALIWG). The question was raised at the Nov meeting as to whether the lack of an APPEND capability in TEXT_IO and SEQUENTIAL_IO was a serious problem for users of Ada. Those in attendance voted overwhelmingly that it was. Several approaches to solving the problem were presented, but the group split almost evenly between two, which were significantly different, as described below. Those present at the January meeting voted to recommend Solution One, below.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

To achieve absolute machine portability, one must open a new version of the output file, copy the entire previous contents of the file, and then continue outputting the data to be appended. This is, of course, unacceptably inefficient.

The alternative is to attempt to encapsulate the I/O operations in an implementation-dependent package and attempt to gain access

to the system file append using the FORM parameter, Pragma
Interface, or whatever means the particular compiler writers have made available.

POSSIBLE SOLUTIONS:

Solution One:

This approach involves adding a new procedure to TEXT_IO and SEQUENTIAL_IO of the form:

```
procedure APPEND      (FILE  : in out FILE_TYPE;
                       NAME : in     STRING;
                       FORM : in     STRING := "");
```

APPEND has the effect of opening FILE so that all previous contents are retained and subsequent calls to PUT or WRITE add their information at the end of the file. Note that there is no MODE parameter - APPEND assumes OUT_FILE. There was some debate as to whether the APPEND should automatically create FILE if it does not already exist (a "friendly" APPEND), or whether it should fail ("unfriendly"). Sentiment was for the unfriendly version. Sentiment also favored a "clean" function for determining whether a named file already exists

-- this is discussed in another comment.

The suggestion that a boolean parameter be added to indicate friendly or unfriendly opens was rejected. The possibility of adding yet another, friendly version of APPEND was left open.

Some participants voiced concern with the behavior of APPEND running on computers that support multiple versions of files within the host file system (such as A VAX/VMS). The final consensus was that it did not really matter how the APPEND procedure was implemented, from a purely Ada point of view. It was pointed out, however, that a reasonable compiler vendor would provide a way for the programmer to achieve the desired results through the FORM parameter.

The biggest advantage to Solution One is that it is upward compatible in nature, that is, not a single line of Ada code already written would have to be modified. It adds the new capability without forcing massive changes to existing programs.

Solution Two

The second solution is to use the existing OPEN procedure but change the enumerated type FILE_MODE to include elements for all possible file interfaces.

A sample of the idea is given below:

```
TYPE FILE_MODE is (     IN_FILE, OUT_FILE, CREATE_FILE, FRESH_FILE,
                        FRIENDLY_APPEND, UNFRIENDLY_APPEND);

-- IN_FILE              open for reading, signal an error if the file doesn't already exist
--
-- OUT_FILE             open for writing, signal an error if the file doesn't already exist
--
-- CREATE_FILE          open for writing, signal an error if the file already exists
--
-- FRESH_FILE           open for writing, delete the file if it already exists (friendly CREATE)
--
-- FRIENDLY_APPEND      open for writing, retain old contents if the file already exists, otherwise
--                      create a new file
--
-- UNFRIENDLY_APPEND    open for writing, retain old contents if the file already exists, otherwise
--                      signal an error
```

Notice that there is no need for the CREATE procedure under this scheme, since creating a file is taken care of by one module explicitly and two others implicitly.

This scheme is very attractive because of the use of a single OPEN and is probably the way file interfacing should have been taken care of in the initial definition of the language. Its one major drawback is that it is a departure from the current file model and adopting it will cause much existing code to be changed.

CONCLUSION:

The lack of an append facility is a significant drawback to Ada, felt by users who need a file-append capability and have no practical, portable way of getting it.

EXTENDING SEQUENTIAL FILES

DATE: September 27, 1989

NAME: K. BUEHRER

ADDRESS: ESI
Contraves AG
8052 Zuerich
Switzerland

TELEPHONE: (011 41) 1306 33 17

ANSI/MIL-STD-1815A REFERENCE: 14.2.1, 14.3.1

PROBLEM:

Extending an external sequential file is a very common file management operation. However, none of the predefined input/output packages (sequential_io , text_io) provides a direct way to open file for extension. The only way (allowed by Ada) to extend an existing sequential or text file, is to create a new file and copy toe contents of the existing file. This is certainly not the most efficient way to do it. An alternative would be to use the form parameter of the open procedure, and pass a directive to the underlying operating system. But the form parameter is implementation-dependent, and the program thereby becomes non-portable - even if file extension were supported by all operating systems!

IMPORTANCE: IMPORTANT

File extension is a common operation, supported by most operating systems. Ada should thus provide a uniform, predefined way to open existing sequential files for extension. Non-portable, implementation-dependent solutions will otherwise evolve.

CURRENT WORKAROUNDS:

Non-portable solutions exist on most implementations.

POSSIBLE SOLUTIONS:

A simple and upward compatible solution is, to add a procedure

```
procedure extend  (file  : IN OUT file_type;
                   mode  : IN file_mode := out_file;
                   name  : IN string;
                   form  : IN string := "");
```

to each of the predefined input/output packages. The name of the procedure may of course be different, e.g. "append". No negative impact on existing implementations can be expected.

CONSISTENT USE OF UPPERCASE AND LOWERCASE

DATE: October 20, 1989

NAME: John Walker

ADDRESS: IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

TELEPHONE: (703) 685-1477

ANSI/MIL-STD-1815A REFERENCE: 14.2.3, 14.2.5

PROBLEM:

Conventionally, and in the LRM, the case of a term indicates whether the term refers to a general idea or to a specific item of software. "Text IO" would presumably refer to the input and output of text; "TEXT_IO" would refer to a particular item of software. At some places, however, the manual's general convention is not followed. (See 14.2.3 and 14.2.5.) At others, it is not clear whether it is being followed or not. For instance, 14.3's title ("Text Input-Output") would indicate that discusses text IO in general; but the first sentence indicates it discusses only TEXT_IO. In the context of the LRM, the two may be equivalent, but the casing in the heading implies that the section will cover a wider topic than just TEXT_IO.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

One simply checks the index to see whether the general topic of text IO is ever discussed apart from TEXT_IO. It's no great difficulty, but it does take a little time.

POSSIBLE SOLUTIONS:

Use the same casing in section headings that is used in code listings.

ADDITIONS TO TEXT_IO

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: 14.3

PROBLEM:

Two subprograms would be useful additions to Text_IO. These are an Exists function and an Append procedure. Currently, the only way to test for the existence of a file is to try to open it and test for Name_Error. This violates the general rule that exceptions should not be used for normal processing. An Exists function would circumvent this necessity. Likewise, to append one file to another is currently very awkward. It involves reading both files into memory and writing them both back to the file to be appended to. This could be done much more efficiently by an Append procedure in Text_IO.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Test for Name_Error to determine the existence of a file. Append by reading both files into memory and writing them back as one file.

POSSIBLE SOLUTIONS:

ADDITIONAL PUT_LINE DEFINITIONS DESIRABLE

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 14.3

PROBLEM:

The predefined procedure PUT_LINE is only defined for string arguments. Users can put character, integer, real or enumeration type values; however, calls to PUT_LINE with any argument other than a string results in a compile-time error. In other words, the argument types accepted by PUT are not the same as those accepted by PUT_LINE. This inconsistency has resulted in a good deal of confusion in the introductory Ada classes I teach. Worse, the typical circumstance is that this inconsistency evidences itself while I am trying to explain a more important concept. Instead of being able to concentrate on the more important concept, we need to also explain that PUT_LINE is not defined for anything other than STRING arguments. This detour is not one that is really necessary. Since we typically deal with this issue concurrently with the need to instantiate generic I/O packages, the net effect of this inconsistency between PUT and PUT_LINE further elevates an introductory student's frustration levels. By defining PUT_LINE procedures to be consistent with PUT procedures, both instructors' and students' efforts can be simplified.

(About the only honest response to "Why can't I PUT_LINE an integer?" "Because that's the way the language is defined.", which is an artificial reason for an artificial situation.)

Note that once you adopt this change, then you also need to extend GET_LINE to be consistent with PUT_LINE in terms of arguments. The amount of work involved to do this appears to be trivial. In fact, the amount of work needed to extend TEXT_IO for more PUT_LINE procedures appears to be trivial, certainly small compared to the benefit to be gained.

IMPORTANCE:

You'll improve students' opinions of the language and make it more logically self-consistent.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Add definitions of PUT_LINE (and GET_LINE) for argument types integer, real, enumeration and character.

MAKE TEXT_IO, SEQUENTIAL_IO AND DIRECT_IO PACKAGES OPTIONAL FOR CERTAIN IMPLENTATIONS

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 14.3, 14.2

PROBLEM:

Not all applications have a need for the Text_IO, Sequential_IO and Direct_IO packages. Embedded applications may not be doing any file I/O at all, or, may be doing it in such a device-dependent manner as to make use of these packages impossible. For applications and environments such as these the implementation of the above packages should be optional. That the implementation is optional should be explicitly stated within the standard.

We are not suggesting that these packages should be optional for all implementations, only for those for which it makes sense to omit them. Certainly, it is reasonable to expect tools used for software development to support input/output facilities. But not all environments are used for software development or for other applications that require file I/O support. Please recognize that the difference should be acknowledged and supported by the language.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Just don't implement them or raise USE_ERROR or some other exception on any attempt to use these packages.

POSSIBLE SOLUTIONS:

The possible workarounds have inherent difficulties. Raising an exception occurs at execution time and it requires that an implementor had to spend some time writing the code to raise the exception. We could save the implementors of such systems time by just allowing them to not implement the packages. And we as developers would be aware at compilation time instead of execution time of any attempt to use these packages. Earlier detection allows for easier correction. Compiler vendors who do not implement those packages run the risk of incurring problems with the validation of their product in the future.

The solution is to include in the standard a statement that these packages are optional for environments or applications which do not require file I/O support. Require implementations that choose to not include these packages to include notification to that effect in the system documentation delivered to customers and potential customers.

If the only reason for these capabilities are for validation purposes, then provide a statement that requires the vendor to provide the required IO interface in Appendix F, moving the items out of SYSTEM, but does not have to form any part of a cross compiler system. It is unreasonable to require a developer of a target environment to interface to a host package to perform operations on non-existing peripherals--just for host testing.

UNPREDICTABLE BEHAVIOR OF TEXT_IO

DATE: April 4, 1989

NAME: Daniel Wengelin (Endorsed by Ada-Europe Ada9X Working Group)

ADDRESS: Swedish Defense Research Establishment
P.O.Box 27 322
S-102 54 STOCKHOLM
Sweden

TELEPHONE: +46 8 663 15 00
+46 8 667 55 59(fax)
E-mail DANIEL@LAPSE.UU.SE

ANSI/MIL-STD-1815A REFERENCE: 14.3

PROBLEM:

The behavior of some subprograms in TEXT_IO differs between different validated compilers. The problem is well known to many Ada users, using more than one compiler.

Example Terminal I/O. See also UI 0038

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Trial and error.

POSSIBLE SOLUTIONS:

A stronger definition of TEXT_IO, giving less room for interpretations, and more emphasis on TEXT_IO in the validation suite.

Effects on existing programs: Minor

ADD GENERIC FORMALS FOR DEFAULT VALUES TO TEXT_IO PACKAGES

DATE: October 23, 1989

NAME: Allan R. Klumpp

ADDRESS: Jet Propulsion Laboratory
4800 Oak Grove Drive
Mail Stop 301-125L
Pasadena, CAL 91109

TELEPHONE: (818) 354-3892
FTS 792-3892
Internet: KLUMPP@JPLGP.JPL.NASA.GOV
Telemail: KLUMPP/J.P.L.

ANSI/MIL-STD-1815A REFERENCE: 14.3

ALIWG ACTION: Favorable vote 1987 in Boston and 1988 in Charleston, W.V.

PROBLEM:

There are four generic packages nested within package TEXT_IO. These are INTEGER_IO, FLOAT_IO, FIXED_IO, and ENUMERATION_IO. Each nested package has several default parameters, e.g., DEFAULT_WIDTH and DEFAULT_SETTING. The problem is that it is impossible to override any default when instantiating a nested package. Therefore, in many cases, a user must either assign a new value to the default or override a default repetitively with every call of a given function.

Example

To print four scalars of type FLOAT on one line, leaving one trailing space per scalar, each user must code either:

```
DEFAULT_FORE := 3;
PUT(A); PUT(B); PUT(C); PUT(D); NEW_LINE;
```

or

```
PUT(A, 3); PUT(B, 3); PUT(C, 3); PUT(D, 3); NEW_LINE;
```

Without the "3", the default (2) would leave no trailing spaces.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

There are no workarounds. The user must write the extra code.

POSSIBLE SOLUTIONS:

The proposal is the same for each of the four nested packages. Using FLOAT_IO as an example, the beginning lines of the specification should read:

```
generic

        type NUM is digits <>;

        FORE : FIELD := 2;
        AFT  : FIELD := NUM'DIGITS-1;
        EXP  : FIELD := 3;

package FLOAT_IO is

        DEFAULT_FORE   : FIELD := FORE;
        DEFAULT_AFT    : FIELD := AFT;
        DEFAULT_EXP    : FIELD := EXP;
```

The generic defaults enable a user to override any default at the time the generic package is instantiated. Declaring nongeneric defaults within the package specification restores visibility via USE clauses referring to the instantiated package. Thus the defaults can be referenced or assigned to. No existing applications software need be changed.

ADD FUNCTION DEVICE_LINE_LENGTH TO TEXT_IO

DATE: October 23, 1989

NAME: Allan R. Klumpp

ADDRESS: Jet Propulsion Laboratory
4800 Oak Grove Drive
Mail Stop 301-125L
Pasadena, CAL 91109

TELEPHONE: (818) 354-3892
FTS 792-3892
Internet: KLUMPP@JPLGP.JPL.NASA.GOV
Telemail: KLUMPP/J.P.L.

ANSI/MIL-STD-1815A REFERENCE: 14.3

ALIWG ACTION: Favorable vote 1987 in Boston and 1988 in Charleston, W.V.

PROBLEM:

There seems to be no way independent of the implementation to determine the line length of an input or output device. For example, one might want to write on a printer or a display a full-width string enclosed in quotations and followed by the catenation character "&".

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Device line length can be ascertained by code such as:

```
LINE_LENGTH_STANDARD_OUTPUT:
                    COUNT;    --Line length of display terminal

FOR I IN 1 .. COUNT'LAST LOOP
      BEGIN
            SET_LINE_LENGTH(STANDARD_OUTPUT, I);
      EXCEPTION WHEN USE_ERROR =>
            LINE_LENGTH_STANDARD_OUTPUT := I - 1;
             EXIT;
      END;
END LOOP;
```

The fact that the workaround compiles, executes, and produces the correct result proves that the device line length is known to the system.

POSSIBLE SOLUTIONS:

Add to TEXT_IO the functions:

```
FUNCTION DEVICE_LINE_LENGTH(FILE: IN FILE_TYPE) RETURN COUNT;
FUNCTION DEVICE_LINE_LENGTH RETURN COUNT;
```

When FILE points to a device of limited line length, or, in the second form when the default is such a device, the function returns the device line length. Otherwise the function returns zero.

PRESERVING AND RESTORING THE CURRENT FILE

DATE: October 11, 1989

NAME: J G P Barnes (endorsed by Ada (UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: 14.3.2

PROBLEM:

The current text files are an important property of the environment in a program at any instance. It ought to be possible to preserve the existing default file at any point, set it to the required file for local processing and then finally restore the originally preserved value so that this aspect of the environment is preserved (such restoration might also occur in an exception handler).

The existence of the function CURRENT_OUTPUT suggests that the language designers had such a use in mind.

The obvious approach might be

```
procedure ACTION( ...) is
        (OLD_FILE: FILE_TYPE:=CURRENT_OUTPUT; -- preserve current default
        file
begin
        SET_OUTPUT(NEW_FILE); --set the current file as required
        ... -- general processing
        ...
        SET_OUTPUT(OLD_FILE); -- Restore original default
exception
        when others =>
        SET_OUTPUT(OLD_FILE); --restore if exception(last wishes)
        raise; --reraise if necessary
end ACTION;
```

Unfortunately this obvious approach does not work because the type FILE_TYPE is (very properly) limited private and so the old value cannot be assigned.

IMPORTANCE: ADMINISTRATIVE

More embarrassing than important--might as well get it right though.

CURRENT WORKAROUNDS:

There is a nasty workaround which uses the fact that the parameter mechanism is not assignment. Thus we can preserve the old value in a parameter of ACTION thus

```
procedure ACTION(OLD_FILE: FILE_TYPE; ...) is
begin
        SET_OUTPUT(NEW_FILE); -- set the current file as required
        ... -- general processing
        ...
        SET_OUTPUT(OLD_FILE): -- restore original default
exception
        when others =>
                SET_OUTPUT(OLD_FILE); --restore if exception (last wishes)
                raise; --reraise if necessary
end action;
```

and then we call ACTION with the function CURRENT_OUTPUT as a parameter!

```
ACTION(CURRENT_OUTPUT, ...);
```

This is very bizarre and is not at all satisfactory; it "feels" as if there must be something wrong with the limited type mechanism altogether because we seem to have bypassed the ability to prevent copying of a limited type.

POSSIBLE SOLUTIONS:

A possible solution is that the function CURRENT_OUTPUT should have been a procedure

```
procedure GET_CURRENT_OUTPUT(FILE_TYPE);
```

which assigned the file to the parameter passed. The first few lines of our procedure could then have been

```
procedure ACTION(...) is
        OLD_FILE:  FILE_TYPE;
begin
        GET_CURRENT_OUTPUT(OLD_FILE):  -- preserve the current file
        SET_OUTPUT(NEW_FILE); --set the current file as required
        ... -- general processing
```

This solution, however, enables the user to copy a file and circumvents the security aimed for through limited types. It might be that limited types cannot be made to work.

Another possible solution might be to introduce a pair of subprograms for preserving and restoring the default state within the package TEXT_IO itself; these would have to work in a stack manner so that their use could be nested.

FUNCTIONAL TEXT_IO.GET_LINE

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am--9pm]

ANSI/MIL-STD-1815A REFERENCE: 14.3.5, 14.3.6

PROBLEM:

Using Text_Io to read strings from text files is currently very cumbersome. The programmer has to establish a string buffer of fixed size (and has to guess what the longest line in the file is going to be), maintain the line length in a separate variable, slice the correct amount of string from the buffer for processing, and so forth.

A functional Get_Line would allow a much cleaner solution to text input:

```
while not Text_Io.End_Of_File (The_File) loop
        declare
                Current_Line : constant String :=
                        Text_Io.Get_Line (The_File);
        begin
                [process the line]
        end;
end loop;
```

IMPORTANCE: ESSENTIAL

String I/O from text files is one of the very worst aspects to using Ada and it is one of the very first things beginning Ada students need to learn how to do: this tends to discourage students before they really learn anything about the language.[1]

CURRENT WORKAROUNDS:

It is possible to implement a functional Get_Line recursively, by reading a single character per frame and

[1]This issue is addressed in a related Ada 9X revision request.

then recursing on the remainder of the line. This is grossly inefficient, but it does work.

POSSIBLE SOLUTIONS:

Add a functional Get_Line to Text_Io. In order to do this, the Text_Io.File_Type will have to be changed from a limited private type to a private type, since functions can only have parameters of mode in and limited private types can only be assigned to parameters of mode in out.[2] The slight loss of safety this introduces is more than offset by the increase in usability of the Text_Io package.

Note: It might be desirable to add similar functions to the Direct_Io and Sequential_Io packages.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will re-compile successfully and will behave identically during execution except for possible small changes in execution speed.

[2]They complain, quite rightly, that nothing this simple should be so hard to do, and that it is much simpler in other languages.

THE DIFFICULTY OF READING A LINE IN A PADDED STRING

DATE: July 4, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 14.3.6

PROBLEM:

Reading a line in a padded string is not easy.

This is a problem for Ada students that quickly want to get somewhere in order to avoid losing faith and Ada teachers that want to allow their students to make some small exercise "on their own".

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

The do it yourself"approach.

POSSIBLE SOLUTIONS:

Why not supporting a padding GET_LINE:

```
procedure GET_PADDED (FILE: in FILE_TYPE;
                      ITEM: out STRING;
                      PAD : in CHARACTER :=");
procedure GET_PADDED (ITEM: out STRING;
                      PAD : in CHARACTER :=");
--Truncates current line to ITEM'LENGTH and fills ITEM with the truncated line
--followed by PAD characters.
```

THE SEMANTICS OF GET_LINE ARE DIFFICULT TO USE

DATE: May 16, 1989

NAME: Stef Van Vlierberghe

ADDRESS: S.A. OFFIS N.V.
Wetenschapstr. 10-Bus 5
1040 Brussels
Belgium

TELEPHONE: +32 2 230.75.70

ANSI/MIL-STD-1815A REFERENCE: 14.3.6(13)

PROBLEM:

The semantics of GET_LINE are difficult to use (to say the least)

The manual states:

> Reading stops if the end of line is met, in which case the procedure SKIP_LINE is then called (in effect) with a spacing of one ; reading also stops if the end of the string is met.

The real problem with the specification given is that the caller of GET_LINE has no way of knowing whether the SKIP_LINE was called or not when the input line size is equal to the buffer side.

The basic problem is shown in boldface. If the optional SKIP_LINE would have been omitted, this would not have imposed a problem on the programmer that would have needed it, he would just need to write a subprogram:

```
procedure GET_LINE_SKIP (STREAM : TEXT_IO.FILE_TYPE;
                         ITEM : out STRING;
                         LAST : out NATURAL) is
begin
        GET_LINE(STREAM,ITEM,LAST); --Defined without the part in bold

        if END_OF_LINE (stream)then SKIP_LINE (stream);
        end if;
end;
```

The reverse is not true, a programmer who would have liked GET_LINE without the optional SKIP_LINE invocation is really stuck, this problem is not so easy to spot and forms a real trap for Ada students.

Imagine the programmer that wants to read the current line (and nothing more) without imposing a limit and without reading character by character. He can use GET_LINE and continue reading with successive GET_LINE calls until the LAST returned equals ITEM'LAST, and this will work until the input line has exactly the buffer length, in which case the program will decide to read the rest (and hence wait for input on a next line).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Read character by character.

POSSIBLE SOLUTIONS:

There are 3 possible solutions:

Solution 1: Drop 14.03.06(13):Clean but not upward compatible.

Solution 2: Modify the semantics of GET_LINE slightly:

> Reading stops if the end of the string is met (and LAST will be returned as ITEM'LAST). If before the end of the string is met, the end of the line is met, the procedure SKIP_LINE is called with a spacing of one; and reading also stops (and LAST will not be returned as ITEM'LAST)

Also not upward compatible.

Solution 3: Modify the declaration of GET_LINE slightly:

```
procedure GET_LINE (FILE : in TEXT_IO.FILE_TYPE;
                    ITEM : out STRING;
                    LAST: out NATURAL);
                    SKIP: in BOOLEAN:= TRUE);
procedure GET_LINE (ITEM : out STRING;
                    LAST: out NATURAL;
                    SKIP: in BOOLEAN:= TRUE);
```

and adapt the semantics to :

> Reading stops if the end of line is met, in which case the procedure SKIP_LINE is then called (in effect) with a spacing of one (if SKIP); reading also stops if the end of the string is met.

This solution is almost fully upward compatible (only interferes with compilation units that "with and use" TEXT_IO and another PACKAGE that declares a homograph of the GET_LINE declaration proposed). But this is a problem "with and use" programmers should be used to.

PICTURES

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 14.3.7, 14.3.8

PROBLEM:

Ada does not have PICTURE variables and PICTURE format as does PL/1 and Cobol.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

User written procedures.

POSSIBLE SOLUTIONS:

Add to the TEXT_IO definitions of get and put for integers and reals picture formatting capabilities. The following procedures should be added to all three of the generic packages INTEGER_IO, FLOAT_IO, and FIXED_IO:

```
procedure get( File    : in  File_Type;
               Item    : out Num;
               Picture : in  String );

procedure get( File    : out Num;
               Picture : in  String );

procedure put( File    : in  File_Type;
               Item    : in  Num;
               Picture : in  String );

procedure put( Item    : in  Num;
               Picture : in  String );

procedure get( From    : in  String;
               Item    : out Num;
               Picture : in  String );
```

```
procedure put( To      : out String;
               Item    : in  Num;
               Picture : in  String );
```

The string parameter Picture should specify the appropriate editing picture. CONSTRAINT_ERROR should be raised if the Picture is not valid. Valid picture codes, as in PL/1 and Cobol are:

9	--	a numeric digit is to be printed.
Z	--	a numeric digit is to be printed, but with zero replaced by blank if there are no nonzero digits to the left.
$	--	prints a dollar sign. If several $'s are present, it becomes a floating symbol.
.	--	prints a decimal point
V	--	aligns but does not print a decimal point
,	--	prints a comma if digits were printed to the left, else a blank
-	--	prints a - sign if <0, else blank. May be a floating symbol.
+	--	prints a + sign if >= 0, else blank. May be a floating symbol.
S	--	prints a + or - sign as appropriate. May be a floating symbol.
/	--	inserts a / into the picture.
B	--	causes a blank to be inserted.
CR	--	inserts the letters "CR" if negative, else blank
DB	--	inserts the letters "DB" if negative, else blank
E	--	prints the letter E of the exponent.
K	--	introduces exponent, but does not print a character.
Y	--	digit position, blank when zero, even if not leading
*	--	drifting picture character, causes all leading zeros and fill characters to be replaced by *
T	--	digit position and encoded sign
I	--	digit position and encoded plus sign
R	--	digit position and encoded minus sign
O	--	non-digit position. Just inserts a O. Works like B, /, or :
(n)	--	iteration factor.

The above is clearly somewhat ambiguous. For details, check any PL/1 or Cobol reference manual. In addition, the following should be added:

:	--	inserts a : into the picture. Similar to / but is useful for time units just as / is for date units.
<	--	inserts a (if the number following it is negative, otherwise blank. May be repeated to be used as a drifting character.
>	--	inserts a) if the preceding number is negative, else blank.

PL/1 and Cobol also have character picture editing, but that is less an issue here, since Ada provides string slicing and concatenation which easily does the same thing.

PUTINTEGER

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 14.3.7(9-12, 16-17)

PROBLEM:

There is a restricted set of formats allowed in the output of a number. This should be extended as described below.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

User must write a procedure.

POSSIBLE SOLUTIONS:

Add the following parameters to both PUT's of 14.3.7(9,16):

```
Item     :  in Num; --already there.
Width    :  in Field := Default_Width; -- already there.
Base     :  in Number_Base := Default_Base; -- already there
Digits   :  in Field := 1;
Sign     :  in Boolean := false;
Radix    :  in Boolean := true;
Unsigned :  in Boolean := false;
```

The meanings of the new parameters should be as follows:

Digits specifies the minimum number of digits to print, With leading zeros supplied as necessary. For example, if Width = 6, Digits = 5, and Item = 92, the output would be " 00092". Sign would always output a sign, either '+', or '-', as appropriate. Zero would be +0. Radix if true would use the current format and if false would suppress the radix markers to simplify other-radix output. Example, if Item = 92, Width = 4, Base = 16, Digits = 4, Sign = false, and Radix = false, the output would be "005C".

If Unsigned is true, the sign bit should be ignored and treated as any other bit. This would allow printing of complete numbers without the sign. This is also obviously machine dependent, but still useful.

WHAT IS THE BEHAVIOR OF TEXT_IO.ENUMERATION_IO OPERATIONS WHEN INSTANTIATED FOR AN INTEGER TYPE

DATE: November 2, 1989

NAME: Edward Colbert

ADDRESS: Absolute Software Co., Inc.
4593 Orchid Dr.
Los Angeles, CA 90043-3320

TELEPHONE: (213) 293-0783

ANSI/MIL-STD-1815A REFERENCE: 14.3.9

PROBLEM:

According to the LRM:

The Enumeration_IO Get procedure "reads an identifier according to the syntax of this lexical element (lower and upper case being considered equivalent), or a character literal according to the syntax of this lexical element (including the apostrophes)." [LRM 14.3.9(6)]

According to the LRM:

The Enumeration_IO Put procedure "Outputs the value of the parameter ITEM as an enumeration literal (either an identifier or a character literal)." [LRM 14.3.9(9)]

Both of these paragraphs presume that the Enumeration_IO package has been instantiated with an enumeration type; however, the actual parameter supplied during instantiation of the package for the generic formal type parameter ENUM can be an integer type. It is not clear whether a developer can use Enueration_IO for any descrete type. For example:

```
with Text_IO;
generic

        type Index_type is (<>);
        type Component_Type is limited private;
        type List_Type is array (Index_Type range <>) of Component_Type;
        with function Image (Value: Component_Type) return String;

procedure Display (List: in List_Type)
is

        package Index_IO is Text_IO.Enumeration_IO (ENUM =>Index_Type);

begin

        for I in List'Range
        loop
```

```
                Index_IO.Put (I)
                Text_IO.Put (ASCII.HT & Image (List (I)));
            end loop;

        end Display;
```

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

This is mainly a portability issue. On some implementations of Ada both Enumeration_IO Put and Get operations will work for both integer and enumeration types. On other implementations one or the other does not work.

POSSIBLE SOLUTIONS:

Clarify the descriptions of Enumeration_IO Put and Get operations with respect to their behavior when the package is instantiated for integer types.

MIXED_CASE

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: 14.3.9(2), 14.3.9(8-9)

PROBLEM:

Capitalization option is not provided in TEXT_IO.ENUMERATION_IO formatting.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Write a user-defined procedure.

POSSIBLE SOLUTIONS:

Change the type definition for TYPE_SET to:

```
type Type_Set is ( Lower_Case, Upper_Case, Mixed_Case );
```

In the Put procedure, Mixed_Case will cause all letters to be lower case except for the first letter and any letter following any non_letter to be upper cased.

BADLY NUMBERED PARAGRAPH

DATE: October 5, 1989

NAME: Elbert Lindsey, Jr.

ADDRESS: BITE, Inc.
1315 Directors Row
Ft. Wayne IN 46808

TELEPHONE: (219) 429-4104

ANSI/MIL-STD-1815A REFERENCE: 14.5

PROBLEM:

At the bottom of page 14-1 in section 14.1 is a paragraph numbered 5. At the top of page 14-2 in section 14.1 is another paragraph numbered 5.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS: Renumber paragraph

DELETE SECTION 14.6, LOW LEVEL INPUT-OUTPUT

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: 14.6

PROBLEM:

This section provides no benefit to the language. Several validated compilers do not even support this package and none suffers from a lack of capability. The fatal flaw of this section is an attempt to define syntax without accompanying semantics. Despite good intentions, it is impossible for a language definition to specify the semantics of this sort of package.

Moreover, the current wording of the section could be interpreted to disallow user defined procedures to accomplish the intended effect of LOW_LEVEL_IO. 14.6:1 says 'Such an operation (i.e., an operation acting on a physical device) is handled by using one of the (overloaded) predefined procedures SEND_CONTROL and RECEIVE_CONTROL. It is reasonable, but certainly not desirable, to interpret the phrase 'is handled by using' to mean exclusivity, that is to forbid applications from setting up their own procedures. This treading on the ground of exclusivity contradicts the advantages gained by providing the opportunity to provide user packages.

IMPORTANCE:

Deleting this section would contribute to the effort to make the standard a more tractable, a more reasonable document to use.

CURRENT WORKAROUNDS:

A better method is for applications to provide their own packages.

POSSIBLE SOLUTIONS:

Delete the section. There are no drawbacks to deleting the section and doing so improves the standard by eliminating potential sources of confusion. Deletion of this section would also make implementing the language more simple by eliminating the need for compiler implementors to figure out what to do with this package. Those implementations that now support the package could continue to do so. Thus, deleting the section benefits many and inconveniences none.

For additional references to Section 14. of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

NUMBER	TITLE	PAGE
0275	USE OF RENAMES	8-51
0367	NATIONAL LANGUAGE CHARACTER SETS	2-11
0369	ADA SUPPORT FOR ANSI/IEEE STD 754	3-103
0392	SEMILIMITED TYPES	7-17
0432	IMPLEMENTATION OPTIONS LEAD TO NON-PORTABILITY AND NON-REUSABILITY (II)	3-124
0438	HANDLING OF LARGE CHARACTER SET IN ADA	2-12
0554	CONSTRAINT CHECKING AFTER UNCHECKED CONVERSION AND IO	13-89
0711	PROBLEMS WITH I/O IN MULTITASKING APPLICATIONS	9-78
0736	INCOMPATIBLE NATIONAL VARIATIONS OF THE ISO STANDARD 646	2-20
0745	INTELLIGENT STRONG TYPING	3-67

SECTION 15. REVISION REQUESTS THAT REFERENCE ANSI/MIL-STD-1815A ANNEX OR APPENDIX

NUMERICAL STANDARDS SHOULD BE PART OF REVISED LANGUAGE

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: New Chapter 15

PROBLEM:

The use of trigonometric functions, square root and others is so common that it should be standardized.

IMPORTANCE: ESSENTIAL

It is essential to many diverse applications that common numerical and mathematical concepts have Ada bindings.

CURRENT WORKAROUNDS:

Many compiler vendors provide some numerical packages. There is a great diversity of syntax and semantics.

POSSIBLE SOLUTIONS:

This is a formal request to determine how additional standards can become part of the Ada language environment. A (proposed) standard such as ISO-IEC/JTC1/SC22/WG9, Proposed Standard for a Generic Package of Elementary Functions for Ada, draft 1.1, could be included as Chapter 15. It could also be a "secondary" standard or a collateral standard. The revision process needs to determine the appropriate guidelines.

RANGE ATTRIBUTE FOR SCALAR TYPE

DATE: January 26, 1989

NAME: R. David Pogge

ADDRESS: Naval Weapons Center
EWTES - Code 6441
China Lake, CA 93555

TELEPHONE: (619) 939-3571
Autovon: 437-3571
E-mail: POGGE@NWC.NAVY.MIL

ANSI/MIL-STD-1815A REFERENCE: Appendix A, paragraph 36.

PROBLEM:

Array objects have FIRST, LAST, and RANGE attributes. Scalar types have FIRST and LAST attributes, but not RANGE. This is inconsistent and mildly confusing to programmers. Consider an enumeration type such as this one.

```
type Suits is (CLUBS, DIAMONDS, HEARTS, SPADES);
```

A programmer is likely to write something like this:

```
for S in Suits'RANGE loop
  ...
end loop;
```

It is natural to expect Suits'RANGE to mean CLUBS..SPADES. After all, if S was a string the best way to write the loop would begin "for I in S'RANGE loop...".

IMPORTANCE: ADMINISTRATIVE

This is ADMINISTRATIVE because it makes the language more consistent and easier to learn.
It is upward compatible because it doesn't nullify existing programs.

CURRENT WORKAROUNDS:

The example problem can just as well be written in either of these two ways:

```
for S in Suits loop
  ...
end loop;

for S in CLUBS..SPADES loop
  ...
end loop;
```

POSSIBLE SOLUTIONS:

Define P'RANGE for a scalar type, or a subtype of a scalar type, in appendix A.

CLARIFY CLASSES OF OBJECTS THAT CAN BE USED AS PREFIXES FOR ATTRIBUTES

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: Annex A, Predefined Language Attributes

PROBLEM:

The standard uses the term 'appropriate for <some type>' in the discussion of several attributes. However, the meaning the term 'appropriate' is meant to convey is never explained. In particular, the attribute P'FIRST is defined (A:14) 'For a prefix P that is appropriate for an array type...'. The entire phrase 'appropriate for an array type' is artificial and devoid of intuitive meaning. Since it's devoid of intuitive meaning, that the phrase is never defined only makes the situation worse. The only way to determine that array objects are suitable for use with 'FIRST is to track down the citations in the standard. The result is that the Annex is of little use as a reference. With the attributes scattered throughout the standard, the chore to determine if an intended construction of an attribute is legal takes entirely too much time.

Interestingly, P'TERMINATED and P'CALLABLE are prefaced similarly 'For a prefix P that is appropriate for a task type:'. In this case using task types, which is the only phrase used in the explanation that resembles a term defined in the standard, is illegal. Task-type 'TERMINATED obviously makes no sense. However, to people learning the language the amount of work required to figure this sort of silliness ou. is daunting. Even those of us who have used the language for some time have difficulty in those areas with which we don't work regularly.

Annex A needs to be rewritten to explicitly state the classes of objects that can be used for the prefix P for each attribute.

IMPORTANCE:

The circular and ambiguous definition utilized regularly within the LRM greatly extends the time and effort required to gain a general understanding of Ada or to determine the answer to a specific question. Poor readability of the standard negatively impacts training and productivity of Ada users.

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

Employing a good technical writer to review the standard for accuracy, clarity and consistency.

REAL_IMAGE

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax,bitnet

ANSI/MIL-STD-1815A REFERENCE: A(18), A(49)

PROBLEM:

Ada does not allow the attributes 'value or 'image to be applied to a floating point or a fixed point type.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Instantiate the generic packages FLOAT_IO and FIXED_IO from TEXT_IO.

POSSIBLE SOLUTIONS:

Change the first paragraph of A(18) and A(49) from:

"For a prefix P that denotes a discrete type or subtype"

to

"For a prefix P that denotes a discrete or numeric type or subtype"

Conversion of real numbers is as important as converting integers and should be as convenient.

UNIFY SOME ATTRIBUTES FOR NUMERIC TYPES

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: Annex A

PROBLEM:

There are some missing attributes and some lack of symmetry that are needed for numerically intensive computing.

Attributes applicable to all scalar types, P

P'FIRST	smallest machine value
P'LAST	largest machine value
P'POS	identity function on integer and real types
P'VAL	identity function on integer and real types
P'SUCC	next larger machine representable value
P'PRED	previous, smaller, machine representable value
P'RANGE	P'FIRST..P'LAST
P'MIN(args)	returns value of type P, the minimum of the arguments
P'MAX(args)	returns value of type P, the maximum of the arguments

Attributes for floating point types P with floating point values X,Y

P'EXPONENT(X)	extract universal integer exponent value from X
P'FRACTION(X)	extract value of type P in range 1/RADIX..1
P'SCALE(X,EXP)	add integer exponent EXP to X's exponent
P'COMPOSE (FRAC,EXP)	compose a machine floating point number
P'SPACING(X)	distance to most remote machine number neighbor
P'rem(X,Y)	the exact remainder of X divided by Y
	floating point numbers with integer values
P'FLOOR(X)	greatest integer less than or equal X
P'CEILING(X)	least integer greater than or equal X
P'NEAREST(X)	integer nearest X, if equidistant return even
P'TRUNCATE(X)	integer nearest X in direction toward zero

Attributes needed for completeness of machine representation

P'MACHINE_EPSILON	machine epsilon
P'MACHINE_SMALL	machine smallest positive value
P'LOGICAL_MANTISSA	number of logical digits in machine representation

P'BASE usable as a type mark

IMPORTANCE: ESSENTIAL

The basic attributes to decompose a floating point machine number, scale a floating point machine number and find the "exact remainder" of a floating point machine number must exists for every target in order to write higher level numerics packages that are portable. Others are in the important class. As a group the set of attributes becomes very important to expand the use of Ada to the numerics community.

CURRENT WORKAROUNDS:

A machine specific package of primitive functions is created for each floating point machine representation for each target computer.

POSSIBLE SOLUTIONS:

Provide attributes in the language or create a standard package of primitive functions. Attributes are preferred because of flexibility in generic units. Details of specification of some of these attributes appear as function specifications in Draft 0.91 of a Proposed Standard for a Generic Package of Primitive Functions for Ada.

MAINTENANCE PRAGMAS

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: B, 2.8

PROBLEM:

Some additional pragmas should exist to maintain software or assist in documentation in support of MIL-STD-483, 2167, 2167A.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None in the language - vendor supplied tools.

POSSIBLE SOLUTIONS:

Pragma Expand-References	:	replaces external references with fully qualified names.
Pragma Pretty_Print	:	formats source according to spacing, indentation, alignment, etc.
Pragma Imports	:	list of "imported" objects, types, units, etc.
Pragma Exports	:	list of "exported" objects, types, units, etc.
Pragma Object-Use	:	list of objects and what unit sets/uses the objects value.

Note: The results of these pragmas only appear in the List file, as with pragma Page and do not effect the source code file.

PRAGMAS LIST AND PAGE SHOULD BE OPTIONAL

DATE: October 23, 1989

NAME: Erhard Ploedereder

ADDRESS: Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

TELEPHONE: (412) 856-3600
E-mail: ploedere@tartan.com
E-mail: ploedere@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: B

PROBLEM:

Pragmas LIST and PAGE are anachronism is this day and age of on-line, screen-oriented program maintenance. Furthermore it contradicts the explicit exclusion of the form or content of listings in the standard (1.1.1(k)).

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Pragmas LIST and PAGE should be downgraded to optional elements of the language.

USE PRAGMA INTERFACE

DATE: October 30, 1989

NAME: William Thomas Wolfe

ADDRESS: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

E-mail: wtwolfe@hubcap.clemson.edu

TELEPHONE: (803) 656-2847

ANSI/MIL-STD-1815A REFERENCE: B.5

PROBLEM:

Pragma Interface can be used with a "language name", but it is not clear that it can also be used with respect to specific versions of a language as well. Moreover, the names of specific language/version/manufacturer combinations are not standardized such that a call to pragma Interface with respect to such a combination will always be portable.

CONSEQUENCES:

If the pragma cannot be used to call Ada 83 from Ada 9X and vice versa, then an important 9X transition technique will have been lost. If different compiler vendors give different names to a single language/version/manufacturer combination, then calls to pragma Interface will not be portable.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Specify clearly in B.4 that pragma Interface operates with respect to language/version/manufacturer combinations where appropriate. Specify a standardized mechanism for building the literal representation of such combinations, such that a call of pragma Interface using a literal representation which was constructed according to that mechanism must be accepted by any compiler which supports pragma Interface with respect to that particular combination. Require that all Ada 83 compilers accept pragma Interface from Ada 83 to Ada 9X (as of the next validation suite), and that all Ada 9X compilers accept pragma Interface from Ada 9X to Ada 83.

NO MEANS TO TURN OPTIMIZATION OFF

DATE: May 1989

NAME: Kit Lester, from material supplied by members of Ada-UK

ADDRESS: Portsmouth Polytechnic
115 Frogmore Lane,
Lovedean,
HAMPSHIRE PO8 9RD
England

TELEPHONE: +44-705-598943 after 1pm EST
E-mail to CLESTER @ AJPO.SEI.SMU.EDU
or to LESTERC @ CSOVAX.PORTSMOUTH.AC.UK
or to C_LESTER @ ARE-PN.MOD.UK

ANSI/MIL-STD-1815A REFERENCE: B(8)

Tracking data: Ada-UK and Ada-Europe 9X group reference UK-011: Endorsed by the Ada-UK and Ada-Europe groups

PROBLEM:

Almost all Ada compilers attempt to optimize, many attempt to highly optimize. The language provides a means to select by which criterion the optimization should be done, but no means to turn optimization off.

However, OPTIMIZERS ARE NOTORIOUSLY THE BUGGIEST PARTS OF COMPILERS. For example, in a survey of Pascal compilers, each optimizing compiler exhibited bugs which led to compiled programs malfunctioning (except when the optimizer could be and was turned off): the most common malfunction was to cause the program to crash!

But IN HIGH-RELIABILITY APPLICATIONS, BUGS ARE SIMPLY NOT ACCEPTABLE even on one-in-a-million probabilities. Consequently the authors of high-reliability code are prepared to accept almost anything to avoid causes of bugs, in particular the high-reliability community is beginning to accept that

* the programming in a high-level language; then
* routinely verifying the target code output from the compiler against the source code is a "belt-and-braces" approach, relative to their more traditional approach of programming in assembler "to avoid compiler bugs" (but probably introducing a higher level of source bugs). The "routine verification" is a formidably labour-intensive task, but that community is prepared to pay such a price, especially given that initially writing in high-level language is less time-consuming than programming in assembler.

An optimizing compiler impedes the belt-and-braces approach in three ways:

* It introduces the already-mentioned bugs, but in a very hard-to-notice manner, because the relationship between the source and the assembler is far from straightforward;

* once a mistranslation has been noticed, it is not easy to correct it by source changes - "patching the produced assembler" is more likely, and is prone to omission after a recompilation;
* minor source changes can lead to massive object changes, so that re-verification can't even capitalize on the labour invested in the prior verification - i.e. making re-verification also a very expensive process.

The high-reliability community might be prepared to live with the last two impediments, but are unwilling in the extreme to live with the first.

At the risk of seeming melodramatic (and where high reliability is concerned, it's easy to seem so, given that one has to envisage worst-case scenarios): to date there have been few fatal accidents in which "computer error" has been implicated, but with the introduction of such applications as fly-by-wire "jumbo" civil aeroplanes, there is an increasing likelihood of such accidents, they are likely to kill more people, and are likely to be highly publicity-worthy. A recent multiple-fatal 737-400 crash in Britain may have been due to "the computer" reporting failure in the wrong one of the two engines, with the result that the pilots killed the healthy engine... Unlikely accidents do happen: it would be unfortunate for Ada if such an accident were traced back to an optimizer-introduced bug. SERIOUSLY-MINDED SOFTWARE ENGINEERS WILL BE VERY CONCERNED WITH SUCH ISSUES IN THE NEXT COUPLE OF DECADES.

There are other (much less crucial) motivations for turning optimization off

* when a program is misbehaving, and a compiler error is expected, the usual strategy is to turn optimization off, and see whether the bug disappears; and if the misbehavior persists, to consider examining the object code;
* during program development, to obtain fast compilation of pre-production test version of the target code;
* for student exercises, again for fast compilation (this is rather insidious in it's effect on the transition to Ada: there are educators who feel they lack the machine resource to support the teaching of Ada: hence the colleges are turning out significantly fewer Ada-trained graduates than might otherwise be the case).

IMPORTANCE:

ESSENTIAL for the high-reliability community;
IMPORTANT for the acceptance and take-up of Ada by other parts of the programming community.

CURRENT WORKAROUNDS:

TO NOT USE ADA

POSSIBLE SOLUTIONS:

To add a third allowed argument to pragma OPTIMIZE, so that as well as the Ada-83 options

pragma OPTIMIZE(SPACE);
and **pragma** OPTIMIZE(TIME);
there would also be **pragma** OPTIMIZE(OFF);

or some such. The semantics of the new option could only be by warm words, but hopefully they would include the point that the new option would be "in order that object code can straightforwardly be verified against the source": that tells implementors the evaluation criterion they should aim for.

Note that some compilers already provide this functionality in non-uniform ways:

GRAPHICS INCLUDED IN SOURCE CODE

DATE: November 1, 1989

NAME: Mary F. Hengstebeck

ADDRESS: General Dynamics
P.O. Box 748
MZ 4048
Fort Worth, TX 76101

TELEPHONE: (817) 935-3011

ANSI/MIL-STD-1815A REFERENCE: Annex B

PROBLEM:

Because design documentation is not always updated adequately, maintenance programmers rely primarily on source code listings. At present such listings do not include valuable design level pictures. If pictures were included into source code, it would be possible to make sure that design information and code was consistent. As the system evolved it would also be possible to print the pictures and stylized headers in a 2167 format, thereby reducing the cost of creating and maintaining a project's design documentation.

Examples of valuable pictures include structure charts: pictures of records, data and control flow diagrams, state transition figures or tables, decision tables, entity relationship diagrams. Now that such pictures are being produced by CASE tools, it would be helpful if they could be included easily into source code listings. Listing options might include:

a. all comments and code printed
b. all comments, pictures, and code printed
c. all comments and pictures printed.

Comments could be further subdivided into top level header type comments versus in line block comments. Maybe only a certain category of comments would be desired at a given point in time.

Now that the workstations are able to easily mix text and graphics, we need to be able to take advantage of the capability.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Editors could use special codes that would indicate to the compiler that the next block of information should be ignored. I do not believe it would be a good idea to force the inclusion of "--" into the left margin of the pictures. If the pictures were going to be printed on standard line printers, the compiler would need to know the graphics standard used to produce the picture, so it could select the appropriate print driver. There would have to be a mechanism for providing this information to the compiler. The

compilers would need to allow more than ASCII in comments, i.e., Pixel images.

SUPPORT FOR UNSIGNED INTEGER TYPES

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: Annex C

PROBLEM:

There has already been an unsigned integer type revision request submitted. We ask that the revision request be given serious consideration.

IMPORTANCE: IMPORTANT

Unsigned integer types would give direct access to underlying integer arithmetic.

CURRENT WORKAROUNDS: See base request.

POSSIBLE SOLUTIONS: See base request.

NO STANDARD SET OF MATHEMATICS FUNCTIONS FOR FLOATING POINT TYPES

DATE: August 9, 1989

NAME: H L Jackson

ADDRESS: Plessey Avionics
Martin Road, Havant,
Hants P09 5HD
Britain

TELEPHONE: +44 705 493203

ANSI/MIL-STD-1815A REFERENCE: C-1

PROBLEM:

The current Ada language standard does not specify a set of mathematical functions for use on floating point types. Other general purpose high order languages invariably include a comprehensive set of math functions such as logarithms, trigonometric functions and their inverses. The omission of these from the Ada language means that portable numerical processing applications cannot be written in Ada.

IMPORTANCE: ESSENTIAL

Ada cannot become a credible general purpose programming language within the scientific/engineering community until a set of math functions becomes a mandatory part of the Ada language.

CURRENT WORKAROUNDS:

Some Ada compilers do include a math library, however, variations between the different implementations hinder the portability of applications using them. A portable math library could be written in Ada thus allowing the math library to be ported along with the application, however, the performance of the math functions would be orders of magnitude slower than the machines own math support.

POSSIBLE SOLUTIONS:

A comprehensive set of math functions must become a mandatory part of the Ada language.

PREDEFINED OPERATORS FOR FIXED POINT TYPES

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: C

PROBLEM:

The following comments concerning fixed point types should be added to package Standard between paragraphs 10 and 11. Currently, information on the operators available for fixed point types is scattered in Sections 3.5.10(14), 4.5.5(6-11) of the Reference Manual.

```
-- In addition, the following operators are predefined for fixed
-- point types:

-- function "="   (LEFT, RIGHT : any_fixed_point_type) return BOOLEAN;
-- function "/="  (LEFT, RIGHT : any_fixed_point_type) return BOOLEAN;
-- function "<"   (LEFT, RIGHT : any_fixed_point_type) return BOOLEAN;
-- function "<="  (LEFT, RIGHT : any_fixed_point_type) return BOOLEAN;
-- function ">"   (LEFT, RIGHT : any_fixed_point_type) return BOOLEAN;
-- function ">="  (LEFT, RIGHT : any_fixed_point_type) return BOOLEAN;

-- function "+"   (RIGHT :      any_fixed_point_type)
--                              return same_fixed_point_type;
-- function "-"   (RIGHT :      any_fixed_point_type)
--                              return same_fixed_point_type;
-- function "abs" (RIGHT :      any_fixed_point_type)
--                              return same_fixed_point_type;

-- function "+"   (LEFT, RIGHT :      any_fixed_point_type)
--                                    return same_fixed_point_type;
-- function "-"   (LEFT, RIGHT :      any_fixed_point_type)
--                                    return same_fixed_point_type;

-- function "*"   (LEFT :       any_fixed_point_type; RIGHT : INTEGER)
--                              return same_fixed_point_type;
-- function "*"   (LEFT : INTEGER; RIGHT : any_fixed_point_type)
--                                         return same_fixed_point_type;
-- function "/"   (LEFT :       any_fixed_point_type; RIGHT : INTEGER)
--                              return same_fixed_point_type;
```

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

MULTIOCTETT CHARACTERS

DATE: March 21, 1989

NAME: Bengt Sundelius

ADDRESS: ABB Automation
dep AUT/KMI
S-721 67 Vasteras
Sweden

TELEPHONE: +46 21 109254

ANSI/MIL-STD-1815A REFERENCE: Appendix C-3

PROBLEM:

There is a need to allow 16 and 32 bits CHARACTERS in some languages in the same way as CHARACTER.

See also AIS 012

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Define LONG_CHARACTER and LONG_LONG_CHARACTER in the same way as CHARACTER. The package TEXT_I0 must be extended in some way. In order not to affect the existing TEXT_I0 it is possible to define a LONG_TEXT_I0 containing I/0 for CHARACTER, LONG_CHARACTER, LONG_LONG_CHARACTER, STRING, LONG_STRING, LONG_LONG_STRING, ETC. An other way is define a new generic TEXT_I0 package which a user could instantiate to the proper CHARACTER type.

DEFINITION OF NATURAL IS INCORRECT

DATE: September 20, 1989

NAME: Michael F. Sargent (Canadian Working Group)

ADDRESS: PRIOR Data Sciences Ltd.
240 Michael Cowpland Drive,
Kanata, Ontario, Canada
K2M 1P6

TELEPHONE: (613) 591-7235

ANSI/MIL-STD-1815A REFERENCE: Appendix C, 16

PROBLEM:

The definition of the subtype NATURAL does not correspond to the mathematical definition of a natural number. The definition of POSITIVE is likewise questionable. This causes considerable confusion amongst those who have learned the correct definitions while in school.

FROM:

Vectors, Matrices and Algebraic Structures, H.A. Elliot, K.D. Fryer, J.C. Gardner, Norman J. Hill, Holt, Rinehart and Winston of Canada Ltd., ISBN 0-03-921054-5:

"N = {all natural numbers} = {1, 2, 3, ...}"

FROM:

Calculus and Analytic Geometry, John A.Tierney, Allyn and Bacon, Inc., ISBN 0-205-04645-2:

"N = {1, 2, 3, ...}"

IMPORTANCE: ESSENTIAL

The standard is incorrect.

CURRENT WORKAROUNDS:

The best current workaround seems to be to disallow the use of the predefined types NATURAL and POSITIVE. Projects should define their own versions of these types which are correct.

POSSIBLE SOLUTIONS:

Change the definitions to:

```
subtype WHOLE  is INTEGER range 0 .. INTEGER'LAST;
```

```
subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;
```

EXTEND TYPE CHARACTER TO A 256-CHARACTER EXTENDED ASCII CHARACTER SET

DATE: September 19, 1989

NAME: Ray A. Robinson, Jr.

ADDRESS: SPARTA
4901 Corporate Drive
Huntsville, AL 35805

TELEPHONE: (205) 837-5282 x1472

ANSI/MIL-STD-1815A REFERENCE: Appendix C (type CHARACTER)

PROBLEM:

Communication and extended ASCII terminal graphics are complicated by the existing CHARACTER type's range (0..127). Greek or graphics symbols and incoming characters with parity bits set cannot be used with this type.

IMPORTANCE: IMPORTANT

This problem requires that 8-bit ASCII data be handled as numbers, which is contrary to the intuitive nature of Ada. Communication software must include utilities to convert CHARACTER strings to and from INTEGER strings.

CURRENT WORKAROUNDS:

Use an unsigned 8-bit integer type when characters may be eight bits wide. Then use implementation-dependent subprograms to read and write the integers as characters.

POSSIBLE SOLUTIONS:

Extend type CHARACTER to a 256-character extended ASCII character set. This might be done by making the existing 128-character type a subtype of a 256-character type with a name like EXTENDED_CHARACTER.

EXPAND USE OF UNIVERSAL INTEGER AND UNIVERSAL REAL

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE: Annex C(5)(8)

PROBLEM:

The predefined types such as INTEGER and FLOAT have several syntactic forms. As a type mark they are integer and float types. As a function they are explicit type conversion functions INTEGER(X) and FLOAT(I). As attribute prefixes they can be functions INTEGER'IMAGE(I) and FLOAT'SAFE_LARGE.

The universal types have implicit conversion. There could be a comment in package STANDARD that reads:

-- function INTEGER (ITEM:universal_integer) return INTEGER;

There could be a definition of INTEGER'SIZE that was stated as -- function INTEGER'SIZE return universal_integer;

The purpose of the syntactic definitions is to point out the problems that happen when a user defines a new numeric type. The fact that the new type is a scalar and is numeric may be transparent to a compiler yet the view is clear to the user.

Consider the possibility of a user defining the following:

```
        type DECIMAL_FLOAT is -- user defined
        type DECIMAL_INTEGER is -- user defined

function INTEGER (ITEM: DECIMAL_INTEGER) return INTEGER;
        -- user provides body
function DECIMAL_FLOAT (ITEM:STRING) return DECIMAL_FLOAT;
     -- user provides body, and has literals ( of sorts )

A : DECIMAL_FLOAT := DECIMAL_FLOAT("-3,245,576.001E+6");

function DECIMAL_FLOAT (ITEM:universal_real) return DECIMAL_FLOAT
        -- user must get compilers rational value and build
        -- a number of the users type

A : DECIMAL_FLOAT := DECIMAL_FLOAT(3.14159);
```

```
function DECIMAL_FLOAT'LAST return DECIMAL_FLOAT;
                -- defining attribute functions
```

The desire is to provide the numerics community with the ability to have user defined numeric scalar types with the normal conversions and attributes. Then these types can become generic actual parameters and act like a numeric type should act.

Other interests might like to define a type such as

```
type US_CURRENCY is -- implementation defined
M : US_CURRENCY := US_CURRENCY ("$7,234,125.50");
                -- and define operations "+" and "*" to operate
                -- according to financial rules with 20 decimal
                -- digit accuracy
```

Numerics has need for arbitrary precision integers for some calculations and for rational numbers (ratio of two arbitrary precision integers) for determining exact results. Numerics has need for arbitrary precision decimal floating point for determining polynomial coefficients and other fixed accuracy computations.

IMPORTANCE: IMPORTANT

A numeric scalar type, even when built from a record with a component being an array, should appear like a numeric scalar type.

CURRENT WORKAROUNDS:

Explicit conversion functions are given slightly different names (thus causing name pollution). Conversions from universal types are handled by double conversion (praying CONSTRAINT_ERROR will not be raised)

```
function CONVERT_DECIMAL_FLOAT (ITEM:INTEGER) return DECIMAL_FLOAT;

A : DECIMAL_FLOAT := CONVERT_DECIMAL_FLOAT( INTEGER(100_000) );
```

POSSIBLE SOLUTIONS:

If there are no significant technical impediments, it seems natural to allow users to define type conversion functions and attributes for their synthesized numeric types.

```
function DECIMAL_FLOAT'LAST return DECIMAL_FLOAT;
                -- defining attribute functions
```

The desire is to provide the numerics community with the ability to have user defined numeric scalar types with the normal conversions and attributes. Then these types can become generic actual parameters and act like a numeric type should act.

Other interests might like to define a type such as

```
type US_CURRENCY is -- implementation defined
M : US_CURRENCY := US_CURRENCY ("$7,234,125.50");
                -- and define operations "+" and "*" to operate
                -- according to financial rules with 20 decimal
                -- digit accuracy
```

Numerics has need for arbitrary precision integers for some calculations and for rational numbers (ratio of two arbitrary precision integers) for determining exact results. Numerics has need for arbitrary precision decimal floating point for determining polynomial coefficients and other fixed accuracy computations.

IMPORTANCE: IMPORTANT

A numeric scalar type, even when built from a record with a component being an array, should appear like a numeric scalar type.

CURRENT WORKAROUNDS:

Explicit conversion functions are given slightly different names (thus causing name pollution). Conversions from universal types are handled by double conversion (praying CONSTRAINT_ERROR will not be raised)

```
function CONVERT_DECIMAL_FLOAT (ITEM:INTEGER) return DECIMAL_FLOAT;

A : DECIMAL_FLOAT := CONVERT_DECIMAL_FLOAT( INTEGER(100_000) );
```

POSSIBLE SOLUTIONS:

If there are no significant technical impediments, it seems natural to allow users to define type conversion functions and attributes for their synthesized numeric types.

For additional references to any of the Annexes or Appendices of ANSI/MIL-STD-1815A, see the following revision request numbers, and revision request titles and pages in this document.

REVISION REQUEST

ADA 9X REVISION REQUESTS
THAT REFERENCE
ANSI/MIL-STD-1815A

SECTION 16. REVISION REQUESTS THAT DO NOT REFERENCE ANSI/MIL-STD-1815A OR REFERENCE THE ENTIRE STANDARD

PROVIDE CHAINING CAPABILITY IN PREDEFINED PROCEDURE

DATE: February 20, 1989

NAME: Chris Clarke

ADDRESS: Berkshire, Fox & Associates
Post Office Box 90673
Pasadena, CA 91109-0673

TELEPHONE:

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Due to the availability of virtual memory, most minicomputer and mainframe programmers rarely consider the size of main memory as a limiting factor when creating their programs. In contrast, the size of main memory is a major concern of microcomputer programmers. The most widely used microcomputer operating systems, MS-DOS, does not have virtual memory capabilities. Without the availability of special programming techniques to get around this limitation, microcomputer programmers would have to severely limit the functionality of their programs, and, it would be impossible to create large, integrated information systems for microcomputers. One of most widely used of these programming techniques is the "chaining" capability provided in many programming languages. "Chaining" gives a programmer the ability to break down large integrated information systems into separate executable programs, and, then, when the system is operated, swap these programs in and out of main memory as the need arises. "Chaining", in effect, simulates virtual memory. Ada does not have the capability to chain programs. As a result, microcomputer programmers who use Ada must severely limit the functionality of their programs.

IMPORTANCE: ADMINISTRATIVE

Microcomputer programmers who use Ada will have to continue limiting the functionality of their programs.

CURRENT WORKAROUNDS:

Programmers must either limit the functionality of their Ada programs or use a proprietary CHAIN command supplied by the compiler manufacturer - which hurts portability.

POSSIBLE SOLUTIONS:

Create a new predefined procedure called CHAIN. When invoked, this new procedure would completely transfer control to the named program and deallocate the memory used by the original program. Preferably, the new procedure would also preserve the common data areas in each program.

REFERENCING PARAGRAPH NUMBERS IN THE REFERENCE MANUAL

DATE: July 25, 1989

NAME: Donald L. Ross

ADDRESS: IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

TELEPHONE: (301) 459-3711

ANSI/MIL-STD-1815A REFERENCE: ALL

PROBLEM:

The Ada Reference Manual would be much easier to use if paragraph as well as section numbers were included in the cross references.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

REFERENCE MANUAL ORGANIZATION

DATE: June 9, 1989

NAME: Barry L. Mowday

ADDRESS: General Dynamics
P.O. Box 748 MZ 5050
Fort Worth, Texas 76101

TELEPHONE: (817) 762-3325

ANSI/MIL-STD-1815A REFERENCE: LRM.all

PROBLEM:

The organization of the standard makes it difficult to read, understand, interpret and, especially, explain. It turns out that not everything in the manual is actually a part of the standard. Annexes A, B, and C are formally a part of the standard; however, appendices D, E and F are not. Yet they all look alike in form. Moreover, the examples and notes within the fourteen chapters are not part of the standard. This distinction unnecessarily complicates the task of a user of the language.

IMPORTANCE: HIGH

CURRENT WORKAROUNDS:

A too-pronounced need to develop language lawyers and resident gurus. Too much effort is required to understand the manual compared to the definitions of other languages.

POSSIBLE SOLUTIONS:

The standard should have the courage of its words. Every statement within the standard should be viewed as being a part of the standard. This does not mean that one should excise all examples, by any means. However, the people responsible for the standard should take an equal degree of responsibility for the examples and notes. Ancillary material can be moved to a separate document that is not part of the standard, if additional discussion of certain points is desirable.

Note that such ancillary documents, the rationale and the implementor's guide, already exist. While it is good to have documents of that sort, make the standard a real standard and excise the non-standard material.

ADA GRAMMAR

DATE: May 15, 1989

NAME: J. A. Edwards

ADDRESS: General Dynamics
P.O. Box 748 MZ 1746
Fort Worth, Texas 76101

TELEPHONE: (817) 763-2612

ANSI/MIL-STD-1815A REFERENCE: LRM.all

PROBLEM:

Currently, to create and mature an Ada compiler, it takes from 3..5 years. For the new architectures of the future and rapid compiler development, the language needs to be expressed in terms that are easy to parse and to generate code.

The definition should be revamped so that the grammar in Ada to conform to LR(m,n) for consistent/complete parsing rules -- the most efficient and accurate compiler techniques. Move more semantics to the grammar specification to rid the language definition of so many special cases.

IMPORTANCE: *VERY HIGH*

One of the major outcomes of the Ada 9x effort.

CURRENT WORKAROUNDS:

Let each vendor provide the language fix-ups and ad hoc parsing for the grammar.

POSSIBLE SOLUTIONS:

<<major impact>> add the rewrite rules to make Ada compatible with LR parsing.

AUTOMATE THE PRODUCTION OF THE NEW LRM

DATE: August 30, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: Entire Document

PROBLEM:

The references at the end of each section of the LRM are incomplete, e.g., 3.7(2) includes

component_subtype_definition ::= subtype_definition

but there is no reference to 3.3.2 for the definition of subtype definition

IMPORTANCE: ADMINISTRATIVE

Otherwise the new LRM will almost certainly have the same sorts of problems as the old one.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Automate the production of the new LRM, especially the syntax definitions and the references, such that all terms used in syntax definitions are automatically included in the references at the end of the section and automatically included in the index. This can be done easily using off-the-shelf software, e.g., macros in UOW Script.

MACHINE-READABLE LRM

DATE: August 30, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: Throughout

PROBLEM:

Setting the Ada manual in type is an expensive process, especially if it is to have correct markup, e.g., bold and italic text. Although vendors are encouraged to reprint the LRM with marked insertions containing implementation information, they frequently chose not to do so due to the expense. Further, this makes it more difficult to develop certain tools for an APSE, e.g., HELP for syntax.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Supply the Ada 9X and subsequent versions of the LRM in machine-readable form, including markup. The markup could be SGML, that of a commercial word-processing program, that of a commercial document formatting program or a markup language specifically designed for the LRM, so long as the encoding of the tags was well documented. Distribute the LRM on all of the media in common use, e.g., 5.25" diskette, 3.5" diskette, 9-track tape and 18-track tape.

ORTHOGONALITY

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: Throughout LRM

PROBLEM:

A large number of ADA constructs are allowed in specific contexts and prohibited in similar contexts. This violates design goals of both Steelman and the LRM, e.g., 1.3(3):

> Concern for the human programmer was also stressed during the design. ... integrated in a consistent and systematic way. ... correspond intuitively to what the users will normally expect.

Lack of orthogonality violates the "law of least astonishment", making Ada more difficult to learn, and in some cases increases the burden on the compiler. It also renders some otherwise useful Ada constructs useless or less useful.

There may be specific cases where allowing a specific construct in a specific context imposes an unacceptable cost even when that construct is not used, but such situations are rare. The vast majority of irregularities of which I am aware can be removed at modest, or even negative, cost, e.g., the first restriction in 7.4(4).

There have already been dozens of revision requests submitted requesting elimination of specific irregularities. Although the majority of them are important in their own right, or essential, e.g., 0088, the general problem must also be addressed.

IMPORTANCE: ESSENTIAL

If these irregularities are not removed, Ada will continue to be hard to learn and hard to debug. Use of Ada will be avoided in favor of more regular languages. Where use of Ada is mandated, costs will be higher. Where the irregularities prevent "information hiding", or force complicated workarounds, programs will be less reliable.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Systematically examine LRM for constructs that are not orthogonal. Where there is not a compelling reason to retain the irregularity, revise the language to allow such constructs in all contexts in which they make sense.

Ensure that new irregularities are not introduced during the revision process. Requests for new functionality should be generalized to allow that functionality wherever it makes sense and has acceptable costs.

STRING MANIPULATION

DATE: September 13, 1989

NAME: Seymour Jerome Metz

DISCLAIMER:

The views expressed in this request are those of the author, and do not necessarily reflect those of Logicon.

ADDRESS: Home: 4963 Oriskany Drive
Annandale, Virginia 22003

Office: Logicon, Inc.
2100 Washington Boulevard
Arlington, Virginia 22204-5704

TELEPHONE: Home: (703) 256-4764
Office: (703) 486-3500 Extension 2295

ANSI/MIL-STD-1815A REFERENCE: Throughout

PROBLEM:

Ada currently has only primitive facilities for text manipulation. The string type is only fixed length, and common operators are not available.

IMPORTANCE: IMPORTANT

The original Ada concentrated on stand-alone embedded systems, in which human interaction was unimportant. Today, Ada is being used increasingly for C&C, MIS and other applications in which large volumes of character data must be manipulated. Without better facilities for manipulating character data, other languages will be used instead of Ada. Where use of Ada is mandated, reliability and portability will suffer, and costs will be greater.

CURRENT WORKAROUNDS:

User-written and vendor-supplied packages. This solution hinders code optimization and causes portability and skill-transfer problems.

POSSIBLE SOLUTIONS:

Define new mechanisms, e.g., operators, for dealing with strings, including varying strings. The facilities in ICON, PL/I and REXX appear to be good models for what is needed. Some or all of these facilities could be in a secondary standard, mandatory except for embedded systems.

SCOPE OF LIBRARY UNITS - CANADIAN 9X

DATE: September 21, 1989

NAME: Stephen Michell (Canadian AWG #006)

ADDRESS: PRIOR Data Sciences Ltd.
240 Michael Cowpland Drive
Kanata, Ontario Canada
K2M 1P6

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

The way that library units are currently specified in the ALRM creates significant difficulty in attempting to develop real applications, particularly ones which try to use "reusable" packages or procedures.

The basic problem is that the Ada context clause, the "with" statement, is used:

1. by the compiler at compilation time to determine a context for the compilation unit, and

2. by the run-time executive at elaboration and execution time to control the execution of the overall program.

These two activities are very different in nature, and having one conte clause mechanism to control both phases places arbitrary and unnatural restrictions on each phase.

The following problems appear to occur because of this problem:

a. Space declared in library units to not recover (except by unchecked_deallocation) because these units never go out of scope.

b. Tasks declared in library units cannot be restarted because they are not in the scope of the main subprogram.

c. Tasks declared in library units cannot terminate via the terminate alternative because they aren't in the scope of the main subprogram.

d. Special features such as "mode shift" are nearly impossible to achieve because developers cannot gain explicit control of all of the components to adjust priorities, etc.

e. Tasks created by allocator, whose scope is in a library package, then terminated, will not have Task Control Blocks recovered because some unit may eventually try a T'TERMINATED which must function correctly. This non-recovery problem eventually may fill the heap and cause catastrophic failures in the program.

IMPORTANCE: ESSENTIAL

This problem seriously effects the way that Ada programs today are designed and developed. When

combined with the artificial restrictions subunits (simple name problem), the problem is magnified again.

CURRENT WORKAROUNDS:

Creating artificial generic units so that they can be created when needed but this has the effect of copying units which often need to be shared.

Extra-lingual solutions, such as modifications to the run-time executive to take explicit control of the environment packages as required.

Create multiple Ada programs which communicate through traditional executive services such as mailboxes or shared memory or files to get more explicit control over the application.

Make extensive use of allocator variables for data objects and task objects so that they can be explicitly terminated or deallocated when needed, i of using preferred Ada scoping rules.

Make extensive use of unsafe programming practices such as unchecked_deallocation to control items which could have been done clea in library packages.

POSSIBLE SOLUTIONS:

1. Permit the "with" context statement to appear immediately before any of the following:

   a. program unit specification or body procedure, function or package

   b. program subunit body - procedure, function, package or task

   c. any "declare" block

2. Separate the compilation "with" (etc., from 1 above).

EXAMPLE:

-- suppose we have a library package called Graphics which we want to call from various places in a program at different times.
-- When we initialize it, Graphics sets up a working area, including possibly tasks to control hardware.

```
package Graphics is

  type supported_device_choices     is ( monochrome_low_res,  monochrome_medium
                                     monochrome_high_res, color_low_res,
                                     color_medium_res, color_high_res );
  type pallet_color_choices         is ( black, cyan, -- and so on white );

procedure open ( chosen_device : supported_device_choices;

-- ... and so on

end graphics;

--
```

```
--      Suppose 4 tasks A, B, C, and D all wish to occasionally use the Graphics package.  When none are
--      using it, it cannot tie up valuable resources.

Package Use_Graphics is
end Use_Graphics;

Package body Use_Graphics is

  Task  A is
  end A;      -- and so on for B, C, and D

end Use_Graphics;
----------------------
separate (Use_Graphics)
with Graphics;
Task Body A is
begin
  loop
    -- ... do stuff
    Begin
      Graphics.open(                                    --      but if we had used
        chosen_device => monochrome_high_res );         --      or if B, C, or D had used it,
                                                        --      it may not be set up correctly.
                                                        --      We really need to be able to force the
                                                                elaboration and
                                                        --      initialization of package Graphics here
                                                                each time we access it.
    End;
  -- Here all vestiges of Graphics should be cleaned up by Ada's scoping rules.
end Use_Graphics;
```

OBSCURITIES CAUSED BY SPECIAL CASES

DATE: August 1, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, UL

TELEPHONE: +44-491-57090

ANSI/MIL-STD-1815A REFERENCE: General

PROBLEM:

There are a number of ad hoc special cases in Ada. Many of these were deliberately introduced into the language in order to make life easier for the programmer. In some cases complexity has been added as a consequence and actually made life harder for the programmer.

It is suggested that a list of all such cases be made and each be considered carefully with a view to removing the feature from the language. It might be argued that the problems with these features have now been identified and means of coping with them established so that their removal is pointless. However, there is a need to make the language as simple as possible whilst retaining functionality; any simplification will bring benefits in compilation time, program debugging and also education

Furthermore, it should be noted that genuine additional functionality is being called for through many revision requests. This will almost inevitably add to the complexity of the language so any compensating simplification will be very useful in keeping the overall language size within reasonable bounds.

Here are some examples of the features concerned.

The ability to omit a package body if none is required. This causes quite awkward problems with recompilation and generics which bring much confusion and irritation to many users. A body should always be required.

The ability to omit a subprogram declaration. This again causes problems with recompilation. In this case, however, the user convenience is high and insisting on a distinct declaration is probably not acceptable. However, and on the other hand it might be worth reconsidering whether a combined declaration and body should not also be allowed for generic subprograms. The present situation is a little odd.

The ability to omit range constraints when declaring constant arrays and discriminant constraints when declaring constant records. This can backfire badly in the array case because the rules regarding the default bounds do not always give the naively expected result. The feature was introduced into the language largely to simplify the declaration of constant objects of type STRING. This may not have been worthwhile since Ada strings do not provide the flexibility required by many users anyway and the ability to omit the range is a poor substitute for a proper solution.

The ability to omit the index number from the attributes FIRST, LAST, LENGTH and RANGE thereby

giving 1 by default for a multidimensional array. The index number should always be given for multidimensional arrays and never for one-dimensional arrays.

IMPORTANCE: IMPORTANT

Some of these sorts of points are quite important, others are merely an untidy nuisance. Of the examples given, the first is probably the most worthwhile.

CURRENT WORKAROUNDS: Not applicable.

POSSIBLE SOLUTIONS:

In the case of always requiring a package body it would be nice to use a null declaration thus

```
package body P is
    null;
end P;
```

or maybe even better

```
package body P is null;
```

which echoes the notation when declaring a body stub.

Solutions for the other examples are obvious.

REMOVING USELESS COMPLEXITY

DATE: August 1, 1989

NAME: J G P Barnes (endorsed by Ada UK)

ADDRESS: Alsys Ltd
Newtown Road
Henley-on-Thames
Oxon, RG9 1EN, UK

TELEPHONE: +44-491-579090

ANSI/MIL-STD-1815A REFERENCE: General

PROBLEM:

There are a number of awkward corners in the language often, caused by unexpected interactions between features which cause compilation problems and yet are of little if any benefit to the user. All they do is mean that the user has to put up with a bigger, possibly slower and more error prone and maybe costlier compiler.

Such corners should be sought out and removed. A list of potential candidates should be developed in collaboration with all compiler developers who are those most knowledgeable in this area; the AVO should also be consulted since they have diligently put tests for such extreme features in the ACVC. The list might then be circulated to a selection of users to check that the features are indeed useless.

Here is an example of such a feature. The ability of a function to return a task object declared in the task. Such a task is, of necessity, terminated before the return but a wholly useless set tricks are required to implement this. See AI-00167.

Note that such corner removal may appear to add further nonorthogonality. It may be that a closer study of individual problems will indicate underlying flaws in the description and composition of language concepts which when corrected cause the problem to vanish.

IMPORTANCE: IMPORTANT

It is important that implementers concentrate on doing what is good for the users and not divert resources getting around useless obscurities. Faster and better implementation will emerge as a result.

CURRENT WORKAROUNDS:

Not applicable

POSSIBLE SOLUTIONS:

Not applicable.

REFERENCE:

See AI-00167 regarding returning a task object.

A SECONDARY STANDARD, LOW-ADA, FOR PROGRAM VALIDATION

DATE: August 16, 1989

NAME: B. A. Wichmann, and endorsed by Ada UK

ADDRESS: National Physical Laboratory
Teddington, Middlesex
TW11 OLW. UK

TELEPHONE: +44 1 943 6076 (direct)
+44 1 977 3222 (messages)
+44 1 977 7091 (fax)
(E-Mail address: baw@seg.npl.co.uk)

ANSI/MIL-STD-1815A REFERENCE: Entire Standard

PROBLEM:

An Ada compiler is a big program and in the foreseeable future, will contain significant bugs. It is therefore difficult to write software with proven reliability in Ada since such software must be compiled with a compiler which will contain bugs. This is unacceptable for some highly critical applications.

Even for applications which are not so critical, the absence of specific faults may be vital. For such areas, it is important to have analysis tools which will demonstrate the absence of these vital faults. The construction of such tools is currently prohibitively expensive since they are more complex than an Ada compiler.

IMPORTANCE:

Very important for some application areas for which Ada is the preferred language.

CURRENT WORKAROUNDS:

The conventional approach, adopted by AVA, SPARK and SAFE Ada, is to define a subset of Ada which can be analyzed by special tools. The problem here is that any subset which is simple enough to allow the production of good tools is likely to exclude most Ada code (thus preventing the use of library software). In any case, since compilers only accept the full language, the problem of the reliability of the Ada compiler is not addressed.

Subsets do not allow the analysis of programs written in the full language. A large application written in the full language cannot be economically converted to a subset (suitable for analysis tools).

Hence we conclude that the existing subset approach only handles a small fraction of the requirement.

POSSIBLE SOLUTIONS:

It is proposed that an additional secondary Standard for Ada 9X be developed along the lines of Low-Ada, as in NPL Report 144/89. The full details are omitted here. The following points should be noted:

1. The approach would be a relatively small addition to the provision of a formally defined static semantics proposed in the Destin workshop.

2. The modification of a conventional Ada compiler to generate Low-Ada should not be too difficult.

3. Such a secondary Standard allows for the competitive supply of Ad validation tools without having to write new Ada source-text analysis tools.

SGML FORMAT

DATE: October 20, 1989

NAME: Bradley A. Ross

ADDRESS: 705 General Scott Road
King of Prussia, PA 19406

TELEPHONE: (215) 337-9805
E-mail: ROSS@SDEVAX.GE.COM

ANSI/MIL-STD-1815A REFERENCE: General

PROBLEM:

The SGML (Standard Generalized Markup Language) is the international standard for the electronic storage of documentation. This standard is currently referenced in the CALS (Computer Aided Acquisition and Logistics System) program by the DOD and by the National Institute of Standards and Technology (NIST) as a Federal Information Processing Standard (FIPS) as well as by other standards organizations. The Ada documentation is not currently available in an

SGML format, and attempts should be made to make it and all supporting

documentation available in such a format. Such a format should also be extended to vendor literature where appropriate.

This would also make it easier to supply Ada literature in other formats such as "pageless technical manuals" (hypertext).

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS: Not applicable

POSSIBLE SOLUTIONS:

Produce Document Type Definitions (DTD) for Ada documentation and release Ada documentation in SGML format using this DTD.

CONSISTENT HYPHENATION OF COMPLEX TERMS

DATE: October 20, 1989

NAME: John Walker

ADDRESS: IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

TELEPHONE: (703) 685-1477

ANSI/MIL-STD-1815A REFERENCE: ALL

PROBLEM:

For various reasons, the Language Reference Manual (like most writing on computers) must make heavy use of complex modifiers. A simple example is "floating point type". A problem arises in that complex modifiers frequently produce ambiguity because they violate the word order English assigns to adjectives. In ordinary English usage, for instance, "floating point type" would mean "a point type that is floating"; it would not mean "the type for floating points". So, does "derived type definition" mean "a type definition that has been derived" or "the definition of a derived type"?

The problem of ambiguity becomes more acute as either the knowledge of the readers goes down or the difficulty of the concept described goes up. The possible permutations become more challenging in an example like "generic formal array type declaration".

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

Many readers of the LRM will already be familiar with most of the basic concepts involved. Any ambiguities will be ignored because one of the possible meanings will be known to be true, others false.

For the less skilled user, the most likely workaround will be to ask a more knowledgeable user which of the two or more possible meanings is correct. Alternatively, the manual can be searched for other references where the meaning is made clear. Frequently, the task is dropped.

POSSIBLE SOLUTIONS:

Remove possible ambiguity by always hyphenating complex terms. "Floating point type" would then have to be printed either "floating-point type" or "floating point-type".

According to the rules of English, it would not be necessary to use a hyphen if the point type were floating. And it is not necessary to use a hyphen in referring simply to a floating point. However, some readers may find it "inconsistent" to see "floating point" in one location, and "floating-point" in another. If that is a concern, over-hyphenation would seem to be fully permissible: it may be fussy, but it's not ambiguous.

USING A CLEAR DELIMITER IN SECTION HEADINGS

DATE: October 20, 1989

NAME: John Walker

ADDRESS: IIT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

TELEPHONE: (703) 685-1477

ANSI/MIL-STD-1815A REFERENCE: ALL

PROBLEM:

Section heading currently have no punctuation between the number and the subject specification (e.g., "13.1 Representation Clauses"). (Chapter headings do, a period -- e.g., "5. Statements".)

This produces an ambiguity. In the absence of a delimiter (such as a period, dash, parentheses, etc.), the number takes the form of a modifier. This is the rule in English, and quite common in administrative usage, where we may refer to classes by a number ("503b organizations", "503c organizations").

The question then arises that if there are "13.2 Representation clauses", are there other varieties? Are there "14.7 representation clauses", perhaps?

For an ordinary user of the manual, there is no problem. The positioning and layout make the usage clear. The problem arises when one quotes the manual and includes a section reference. For publications referencing subjects like Ada, the respective manuals are the "official" style guides.

If section headings are referenced as they are printed in the manual, then we have the ambiguity mentioned above; if punctuation is inserted, then there is an inaccurate reference.

IMPORTANCE: ADMINISTRATIVE

CURRENT WORKAROUNDS:

The workarounds are either to permit the ambiguity or to use an inaccurate title as a reference.

POSSIBLE SOLUTIONS:

Use some punctuation after each section number; a comma, period, or dash are common; putting the description in parentheses is also possible.

TIME BOUNDS ON ADA PRIMITIVE OPERATIONS

DATE: October 28, 1989

NAME: Henry G. Baker

ADDRESS: Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436

TELEPHONE: (818) 501-4956
(818) 986-1360 FAX

ANSI/MIL-STD-1815A REFERENCE: ALL

SUMMARY:

As Ada is meant to be used in environments where the timing of certain operations is mission-critical, it would be most helpful if the Ada language specification gave some strong guidance to Ada implementors regarding the speed of certain operations. Since absolute speeds are extremely difficult to prescribe, and since the Ada language standard does not want to unnecessarily constrain implementation techniques, it is difficult to recommend specific requirements or constraints that an implementation should conform to. Nevertheless, some strong recommendations should be included in the document whenever possible. Furthermore, implementors should be required to warn programmers of "screw cases" where the time for an operation could be dramatically longer than usual, because these cases may not show up in normal testing.

SPECIFIC REQUIREMENTS/SOLUTION CRITERIA:

The Ada language standard leaves too much freedom for implementors to implement "sloppy" or "slow" algorithms for certain Ada language features. These implementations are correct in the sense that they produce the correct output, but the algorithms used are not appropriate for mission-critical needs. Obvious examples include the use of the CASE statement -- sloppy implementations are free to implement every case statement as a linear search of possibilities, rather than some more efficient technique such as binary search, hash tables or jump tables. For time-critical tasks, the timing of a statement becomes just as important as the normal semantics of what it computes, because if the correct computation is not performed in a reasonable time, the system will fail, regardless of how correct the computation was in some formal sense.

We request that the Ada standard include strong recommendations to implementors regarding the timing behavior of certain constructs. Programmer intuitively depend on the kind of behavior presented below, and should be warned if this behavior is not to be expected. In other words, the Ada implementation should warn the programmer if there are any "mines" lurking in the implementation which may cause drastically slower behavior. Below are a few examples, but many more could be constructed.

1. The time required to execute a CASE statement on a dense set of cases should be independent of the number of cases. Ada implementors are requested to expect CASE statements having hundreds or thousands of distinct cases.

2. The time required to perform an assignment should be bounded by a constant times the amount of storage affected (as determined by X'SIZE).

3. The time required to allocate storage for an access type should be bounded by a constant times the amount of storage allocated (as determined by X'SIZE).

4. The time required to access an element of an array should be bounded by a constant.

5. The time required to execute a "goto" statement should be bounded by a constant.

6. The time required to execute a procedure entry -- not counting the time to elaborate the procedure body -- should be bounded by a constant times the amount of storage involved in the procedure arguments.

7. The time to elaborate a declaration should be bounded by a constant times the amount of storage affected.

8. The amount of time required to access a component of a non-discriminated record is independent of the size of the record, but may be dependent upon the size of the component.

9. The time required between the raising of an exception and reaching the appropriate handler is bounded by a constant times the number of block levels between the raiser and the handler.

10. The time required to perform an adding arithmetic operation on integers is bounded by a constant times the logarithm of the length of the ranges of integers.

11. The time required to perform a multiplying arithmetic operation integers is bounded by a constant times the square of the logarithm of the length of the ranges of the integers.

JUSTIFICATION/EXAMPLES/WORKAROUNDS:

Timing suites such as Ada performance suites will go a long way towards providing systems designers with estimates of the timing of various Ada language constructs. However, as is usual with published timing benchmarks, there will be enormous pressure on implementors to show good results on the benchmarks, while average and/or worst case behavior is dramatically worse than the implementations where the benchmarks. Some mechanism should be included to push implementors toward robust implementations where the benchmarks will be more or less indicative of the actual performance that can be expected. Some mechanism should be included to warn users of situations where performance can be expected to be dramatically worse than the published benchmarks.

Of course, no amount of documentation or validation of an implementation can eliminate the need for extensive testing of an actual system which utilizes that Ada implementation. However, the Ada implementor should be required to ware the systems designer where performance could be much worse that expected, so that appropriate tests can be devised for those cases.

Steelman requirement 13D can be interpreted as requiring these warnings.

NON-SUPPORT IMPACT:

Increased cost of verification and validation of real-time systems. Increased risk of field failure or real-time systems.

POSSIBLE SOLUTIONS:

Indicated above.

DIFFICULTIES TO BE CONSIDERED: MANY

LACK OF LITERAL REPRESENTATION FOR ABSTRACT DATA TYPES

DATE: August 27, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

It is not possible to define literal representations for user-defined abstract data types.

IMPORTANCE:

CONSEQUENCES:

This combines with the lack of a user-definable := operator to render automatic initialization of user-defined data types impossible, leading to reduced readability and a lack of completeness.

CURRENT WORKAROUNDS:

Assignments must be done in the procedural code rather than at the moment of creation.

POSSIBLE SOLUTIONS:

Design a mechanism by which functions for converting to and from literal representations can be specified for user-defined abstract data types (i.e., limited private types declared within (usually generic) package specifications).

This function should allow user-defined data types to integrate into the compiler's literal representation recognition mechanism, as well as into the I/O system (which presently performs the conversion to literal representations for integer, real, and enumerated types, but which cannot be loaded with the appropriate conversions for user-defined types).

Reference: Abstraction Mechanisms and Language Design (Paul N. Hilfinger's ACM Distinguished Dissertation)

ADA 83 DOES NOT PERMIT THE DEFINITION OF CLASSES OF TYPES

DATE: August 27, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Ada 83 does not permit the definition of classes of types.

IMPORTANCE:

CONSEQUENCES:

Software development is made more difficult.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

A class would consist of a series of procedures and/or functions which are common to any ADT which falls into the class. Then the class can be considered as the set of all ADT definitions which include the enumerated set of operations. Whether or not any ADTs are currently defined which happen to fit into that general class of types isn't of any concern; the number of type definitions actually falling into a given class at any given time could fluctuate continuously.

As a concrete example, assume the class FRUIT has been defined. Then we can write an Ada 9X procedure specification:

```
procedure SLICE (SOME_FRUIT : in out FRUIT);
```

and compile the procedure spec just like it is. Now later on when somebody wants to call this procedure, they can send in an apple, an orange, or some type of fruit which was just discovered yesterday, and as long as the ADT definition of the actual parameter meets the requirement that all the properties of a FRUIT be present, the procedure call will compile perfectly. Since Ada is strongly typed, it is always possible to determine the type of an object at compile time, and so the cost of the mechanism consists only of a

generalization of Ada 83's compile-time checking of parameter types.

The generic mechanism provides a facility whereby the described example could be implemented in Ada 83, but it would require that all the relevant properties of a FRUIT be explicitly listed as generic parameters, which is quite wasteful of programmer time. Additionally, since the generic parameter structures of different procedures are not in any way linked, no recompilation dependencies exist which would enable developers to trace the implications of modifications to a particular class of ADTs; the references to that class are widely dispersed when the generic mechanism is used, and being certain that all implications have been considered is quite difficult when all generic parameter structures must be checked manually.

Generalizations beyond these three levels, objects, types, and classes, do not seem intuitively necessary, since one could define a class which consists of precisely those properties common to several other classes. Then when one writes a procedure requiring a parameter from the new class, any object fitting into any of the classes which were generalized over will be accepted as a valid parameter. This suffers a drawback, though, in that classes whose semantics require a linkage to certain other classes cannot be expressed in such a way as to trigger recompilation if those other classes change, so it may be desirable to also have a synthesize clause, syntactically valid only in the context of a class definition, whereby classes from which properties are to be collected are listed in a manner similar to the use clause provided by Ada 83. The compiler must then enforce a prohibition on the creation of circular dependencies. The APSE should then provide a tool which lists all classes to which a given class also belongs, and one which lists the full specification of all procedures and functions characterizing a given class, including those which are directly or indirectly synthesized from other classes. Whenever a class from which other classes are synthesizing properties changes, the classes synthesizing properties from the modified class must be automatically marked for recompilation. Additionally, it may be desirable to require an automatic analysis of all known classes to determine whether their status with respect to other classes (i.e., those in the transitive closure of the modified class) has changed; if it can be determined that a class is now no longer a member of some other class, or that a class is now a member of some other (recently modified) class, then e-mail should be sent to the owner of the class(es) whose status has changed. The class relationship maintenance requirements will be bounded by the boundaries of an Ada library; the existing mechanisms for linking to specific parts of other Ada libraries or to entire Ada libraries can be used to extend the network of class relationships as necessary.

A class of types would also have the property that it could be used to speed up the definition of new ADTs. For this purpose, an inherit clause, analogous to the Ada 83 use clause, and syntactically valid only in the context of an ADT definition, would be used as follows:

a. ADT Specification

A class definition would first be listed in a with clause, and then listed also in an inherit clause. The effect of the inherit clause is to cause the procedures and functions which characterize the class to be automatically included in the specification of the ADT. There could be more than one class listed in the inherit clause. In the context of a specification, the possibility of conflict between identical procedures in different classes is probably immaterial, because if two given procedure specifications are identical, it does not matter which one is ultimately used.

Clearly, if an ADT specification uses an inherit clause to include the procedures and functions of a given class, then for any procedure requiring a parameter from the specified class, instances of the ADT so defined will meet that procedure's requirements with regard to that parameter. It is also possible that the combination of properties inherited from different classes could cause the ADT to be a member of certain other classes as well, causing its instances to also be acceptable

whenever parameters from those other classes are required.

It would also be possible, though redundant, to give a full specification of a procedure or function which was already included in the ADT specification by virtue of an inherit clause. In that case, the compiler should probably alert the developer of this fact via some informative message.

With regard to recompilation, all ADT specifications having an inherit clause should be automatically marked for recompilation whenever any of the classes from which they inherit have changed. Additionally, all ADTs which are members of a modified class but which did not directly inherit from that class should be automatically analyzed to determine whether or not they still fall into the class, and all other ADTs should be analyzed to determine (in the case of a removal of class characteristics) whether they now fall outside the class. The results of this analysis should be automatically e-mailed to the owners of the ADTs whose class membership status has changed. The APSE should provide tools which list all classes of which a given ADT is a member, whether directly (via an inherit clause) or indirectly (because it contains all the procedures and functions which characterize the class); it should also provide a tool for listing the full specifications of all of the ADT's procedures and functions, including those which are inherited.

b. ADT implementation

Just as it was possible to include procedure and function specifications via an inherit clause, it should also be possible to include implementations which are provided by the classes. Thus, the type classes described above would have both specifications and implementations. The implementation of a class would provide a state representation structure which would be automatically included in the representation of any ADT whose implementation contains an inherit clause for the class, and would also provide the procedure and function implementations of the corresponding portions of the ADT's specification.

Since the possibility of conflict between different implementations of a procedure or function which is provided by more than one class is now significant, the visibility rules of Ada 83 would be used to resolve such a conflict; thus, implementations provided by classes listed later in the inherit clause would obscure implementations provided by classes listed earlier. Implementations provided by the programmer would override any implementations obtained via an inherit clause, and compilers should probably generate an informative message along the lines of "Procedure implementation X from class Y overridden".

Some inefficiency would probably arise whenever this happened, in that there may be a significant amount of descriptor information (etc.) in a class's implementation which would go unused because the procedures which were being supported have been overridden. Thus, perhaps there could also be a mechanism for defining the implementation of only a specific subset of a class's properties; the compiler would then select the smallest subset-implementation which included all of the properties actually being utilized.

Also, it is probably not feasible in many instances to provide an implementation of the entire class; for example, if a COLOR function was a property of the class FRUIT, it might not be reasonable to define a default implementation of that function; only those procedures and functions whose implementation can be reasonably determined by definition of the class's semantics should be encoded into a class's implementation. Thus, class implementations should cover (possibly proper) subsets of a class specification, unlike type implementations, which are required to provide functionality for each and every specified property of the ADT.

ADA 83'S INABILITY TO DEFINE DESTRUCTORS

DATE: August 27, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenuc #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Ada 83 does not support the ability to define destructors very well; although one can write a DESTROY procedure, it cannot be done in such a way as to ensure that when a block is exited, the objects which are local to that block are properly destroyed.

IMPORTANCE:

CONSEQUENCES:

There are certain component users who must maintain complete control of space utilization. Such users are currently forced to either manually invoke DESTROY on each and every object declared in the local environment as the last N statements of every block, or accept the wasted storage which results from the fact that ADTs are typically heavy users of pointers and that none of the space obtained by the ADT to support its representation will ever be properly released.

As a general matter, it is important that the use of storage be precisely defined for every mechanism in the language so that a program's storage requirements can be analyzed using the techniques which are commonly applied to the analysis of time requirements. Unfortunately, Ada 83 rarely if ever addresses space considerations, and shows little concern for the needs of those who wish to explicitly manage space. This is quite distressing, given that all components must explicitly manage their own space if they are to be acceptable to the widest possible class of users, including those who must maintain complete control of space utilization.

CURRENT WORKAROUNDS:

Manual invocation of a DESTROY procedure, which is error prone.

POSSIBLE SOLUTIONS:

Provide a means of defining a destruction procedure for user-defined ADTs such that the destruction procedure is integrated into the block exit mechanism so as to guarantee the automatic destruction of user-defined ADTs upon block exit.

THERE ARE NO STANDARD WAYS FOR ADA PROGRAMS TO COMMUNICATE WITH ONE ANOTHER

DATE: August 11, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: None

PROBLEM:

There are no standard ways for Ada programs to communicate with one another.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Non-portable vendor- or user-specific communications packages.

POSSIBLE SOLUTIONS:

Define standard packages for inter-program communications, or support rendezvous between tasks in different programs. Ideally there would be a standard interface to a lower level user-supplied transport service e.g., TCP.

SPREADING AN ADA PROGRAM OVER MORE THAN ONE PROCESSOR

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

The Ada language is such that spreading an Ada program over more than one processor requires considerable extra effort by the compiler vendor to achieve validation; this effort is wasted because it goes into supporting features of the language (e.g. variables shared between tasks) that would not in practice be used in an arbitrary way in such programs.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Excessive division of system into small programs communicating with each other in a non-standard way.

POSSIBLE SOLUTIONS:

Define permissible implementation-definable limits on visibility between parts of a program running on different processors.

ASYNCHRONOUS INTER-TASK COMMUNICATION NOT AVAILABLE

DATE: August 1, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: None

PROBLEM:

Asynchronous inter-task communication not available.

IMPORTANCE: ESSENTIAL

For many large military systems, asynchronous message passing is the only satisfactory way to handle system concurrency.

CURRENT WORKAROUNDS:

Use extra tasks to pass messages and manage message queues.

POSSIBLE SOLUTIONS:

Provide asynchronous message passing facilities.

Provide support for semaphores.

Provide support for queues (to allow a task to own and manipulate a queue, and for other tasks to put messages on it).

IT IS DIFFICULT TO USE ADA TO WRITE AN OPERATING SYSTEM

DATE: August 11, 1989

NAME: J R Hunt

ADDRESS: Plessey Research
Roke Manor, Romsey
Hants SO51 0ZN
England

TELEPHONE: +44 794 833442
E-mail: jhunt@rokeman.co.uk

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

It is difficult, if not impossible, to use Ada to write an operating system. For example, a multiprocessor naval command and control system might need basic software, comparable in complexity to a minicomputer network operating system, to support fault tolerance, load sharing, change of system operating mode etc. It is highly desirable that such important software be written in Ada, and be portable, i.e. be quite independent of the compiler supplier's Ada run time system. Currently, it would be very difficult to do this in Ada, because of the difficulty of manipulating tasks of arbitrary type and parentage.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use operating systems written in C or assembler.

Write the operating system as an extension of the Ada run time system - this is undesirable because it is non-portable and unvalidated.

POSSIBLE SOLUTIONS:

Support the notion of a task independent of a program. Provide additional ways to refer to a task and to control it.

ADDITIONAL PREDEFINED PACKAGES

DATE: June 15, 1989

NAME: Mike McNair

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

In support of "...large scale and real-time system." (foreword), there is frequently a need to support process control, inter-process communication, and inter-processor communication, where "process" refers to "environment task" of 10.1(8). Inter-processor communication, where "process" refers to "environment task" of 10.1(8).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

None in the language - very run-time environment specific

POSSIBLE SOLUTIONS:

Define package(s) to standardize the interface to services to support process control, inter-process communication, inter-processor communication. There would necessarily need to be Appendix F notes discussing applicable processors/targets, etc.

INHERITANCE

DATE: June 20, 1989

NAME: Mike McNair, Eric C. Aker

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5871, (408) 720-5212

ANSI/MIL-STD-1815A REFERENCE: N/A

PROBLEM:

With the popularity of Object-Oriented techniques and the subsequent desire to utilize these techniques within Ada software, the Ada language must be brought up to date with language features which support an Object-Oriented implementation. In particular, is the concept of 'inheritance' - a notion used to express the commonality of operation(s) along an object tree structure.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

No language support - very cumbersome to achieve and enforce.

POSSIBLE SOLUTIONS:

Since inheritance is concerned with the propagation of a set of operations along an object hierarchy, it is important to pin down the applicable scope and propagation rules. In the scheme outlined below, the propagation of operation names is bottom-up in an object class hierarchy. The scope is dictated by two new pragmas and the encapsulation of a type and operations in a package (object class). The proposed rules are:

1. an object class is defined to be the grouping of a type with a set of operations into a package

2. if the type defined in an object class is an aggregate type, then an object class is created for each of the component types; this is necessarily recursive

3. in order to specify inheritance, the pragma Inherit is placed immediately following the context clause but before the package specification; this pragma names the packages (implementations of object classes) in the context clause whose operation names are to be inherited into the object class

4. an object class which inherits the names of operations from other object classes must supply operations of the same name; this is a compile-time check which results in a compilation error if it fails

5. the type declared within a object class must at least have its name declared within the package specification; as a consequence, an object class type may be either limited, private or 'public'

6. inheritable operations must be declared in the visible part of the object class package specification; it is possible that the same specification will contain operations which are not identified as inheritable

7. those operations whose names can be inherited, must be named as a parameter to pragma Export; this pragma occurs within the package specification of an object class, it can occur more than once and for each occurrence it lists the operation names to be available for inheritance into another object class

8. a generic package may be used to implement an object class; in this case, the generic formal parameters specify implementation details (as is usually the case) and cannot, in the case of generic formal subprogram, be made available for inheritance

9. an object class may inherit operations from one more other object classes

10. because an object class is implemented as an Ada package, the addition of operations or type components (and hence imported types) may effect the compilation and elaboration ordering from the original; Additionally, the compiler must check the inheritance structure based on potentially new relationships

Example:

```
package B is
  type U is ...;

  procedure Op1 (P :  in     out    U);
  pragma Export (Op1);                          -- this pragma specifies that
                                                -- the name Op1 can be inherited
                                                -- by another Object Class

  function Op3 (P :   in            U) return W; -- this operation is not
                                                 -- inheritable, but is available
                                                 -- for objects of type U

end B;

package body B is ... end B;


with B;
pragma Inherit (B);                             -- this pragma specifies that B is
                                                -- a package which is also an
                                                -- object class and which has one
                                                -- or more inheritable operation
                                                -- names

package A is
```

```
   type T is ...;

   procedure Op1 (P :    in      out     T);          -- this operation is required
                                                      -- the name Op1 was inherited from
                                                      -- object class B

   procedure (Op2 (P :           out     T);          -- this is an operation available
                                                      -- for acting on variables of type
                                                      -- T

   function Op3 (P :     in              T) return B.U; -- this is not a definition for an
                                                      -- inherited operation; it just has
                                                      -- the same name

end A;

package body A is ... end A;
```

HOMOGENEOUS DISTRIBUTED MULTI-PROCESSOR SUPPORT

DATE: May 23, 1989

NAME: Charles Corwin

ADDRESS: Link Flight Simulation Division of CAE-Link Corporation
1077 E. Arques Avenue
Sunnyvale, CA 94088-3484

TELEPHONE: (408) 720-5885

ANSI/MIL-STD-1815A REFERENCE: N/A

PROBLEM:

Many embedded Ada Applications REQUIRE a distributed target environment. Since this is not currently within the scope of Ada, application software is being developed which is full of non-portable inter-processor communication software.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Several compiler vendors supply customized inter-processor communication (IPC) packages for a given target machine. A toolkit of "Distributed" components has been developed including a queue, mailbox, semaphore and event flag which hides the distributed nature of the target from the application software.

POSSIBLE SOLUTIONS:

A standard set of packages supporting DISTRIBUTED processing should be made an optional part of the language. As a minimum they should provide a mechanism for passing data between processors while insuring data integrity, provide event signaling and shared resource management. As an alternative solution, Ada could specify inter-processor Rendezvous.

UNSIGNED INTEGER

DATE: April 4, 1989

NAME: Toomas Kaer (Endorsed by Ada-Europe Ada9X working group)

ADDRESS: Ericsson Radar Electronics AB
S-431 84 Molndal
Sweden

TELEPHONE: +46 31-671543
+46 31-276043 (fax)
E-mail: Kaer@MOERE3.ERICSSON.SE
E-mail: Toomas@EUROKOM.IE

ANSI/MIL-STD-1815A-REFERENCE:

PROBLEM:

The following Ada Language issue has been discussed by Ada in the Sweden Ada9X working group. Since there is an ALIWG (LI47) on this issue, the text below is an extract from it. Ada Europe agrees on the problem as such but not necessarily the solution to the problem. The viewpoint of AIS is to highlight the requirement not to solve the problem.

...
LI47 (NEW)

a. Provide unsigned integers

- The basic problem is that Ada requires types to be symmetric. Because negative numbers must be represented, a 32 bit field cannot be used to represent a non-negative integer with 32 bits of precision. There are basically 3 ways to provide for an unsigned integer:

  (1) Syntax language change allowing unsigned integer type

  (2) Pragma allowing for unsigned integers
  (this way is disliked by Ada-Europe)

  (3) Representation clauses providing the capability

An initial issue concerns unsigned literals. Now with a normal machine with a 32 bit integer type, the normal range is from -2**31..(2**31 -1). Hence system.max_int is 2**31 -1. How do you declare a literal bigger that system.max_int? In other words, how do you get an unsigned literal to work correctly? The predefined integer type can be 32 bit signed type, and you can't represent (2**32 -1) in a signed number. Hence you not only must require that the type unsigned, but also that you can have literals of that unsigned type.

A second issue concerns the dealing with constraint checks. Some machines do constraint checks on signed integers on overflow and don't do them on unsigned integers. The problem is you want to know. Either you want that behavior and willing to pay the penalty for the extra instructions or you don't want it. So

we need a mechanism to control this, perhaps some variant of pragma suppress.

A third issue concerns the operations of unsigned integers. As a note, AI361 is currently considering the inclusion of unsigned integer as a representation specification. However this would only allow for the representation of a 32 bit integer field. Arithmetic can be either modulo arithmetic or may result in an overflow when a result is out of range. AI361 would not provide for operations on this representation. What people really seem to want is a type similar to the integer type in all respects without the requirement for a symmetric type. The user would like all the same operations with the standard definitions. However, should the operators have the same semantics? An expression (using an 8 bit unsigned) such as 220+220 could result in either a numeric error or 440 MOD 256. It was pointed out that numeric operators are not defined by the Ada LRM, only that they have their conventional meaning. Definitions for each of the operators must be provided too include "+", "-", "*", "/", rem, mod, "**", abs, and not. Even the treatment of the unary negation operator must be decided (e.g., result a numeric error, result zero). If operations are provided, should there be a mechanism to select a cyclic behavior or an acyclic behavior?

A fourth issue concerns the desirability to support other operators such as relational operators or even logical operators. Further should an unsigned type have attributes of a discrete type?

Justification for unsigned representation is available. What is lacking is justification for unsigned operations. Several usage examples follow: One of the biggest use of unsigned integers was in the handling of 8 bit characters going from 0 to 255. Then numbers could be read in and converted to character using the trivial POS attribute. In general this use does not require arithmetic. Another example is the interface to unix using "inodes" consisting of 32 bit unsigned numbers. A third example is the SUN RPC mechanism where every RPC is registered with a number as a unique identifier. By convention user defined RPC numbers have the high order bit set. The use of the RPC required the addition of a 32 bit unsigned literal. Here there was a requirement to do use both literals and perform arithmetic. The last example provided was the actual representation of an address on a 32 bit machine. Most machines do not have negative addresses. Also most machines start to stack high and work their way down. Without unsigned integers, it is very hard to represent these addresses. Again this is an argument for the unsigned literal...

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

DYNAMIC PRIORITIES

DATE: April 27, 1989

NAME: Toomas Kaer (Endorsed by Ada-Europe Ada9X working group)

ADDRESS: Ericsson Radar Electronics AB
S-431 84 Molndal
Sweden

TELEPHONE: +46 31-671543
+46 31-276043 (fax)
E-mail: Kaer@MOERE3.ERICSSON.SE
E-mail: Toomas@EUROKOM.IE

ANSI/MIL-STD-1815A-REFERENCE:

PROBLEM:

The following Ada Language issue has been discussed by Ada in the Sweden Ada9X working group. Since there is an ALIWG LI (LI48) on this issue the text below is an extract from it. A1s agrees on the problem as such but not necessarily the solution to the problem. The viewpoint of A1S is to highlight the requirement not to solve the problem.

See also AdaUK 012.

LI48

Provide dynamic priorities:

> Embedded real-time systems need a dynamic priority mechanism. Task priorities in Ada are static, but it is possible to work around this fact by taking advantage of the fact that a rendezvous executes at the highest priority of the two participating tasks. It is felt, however, that this is a cumbersome, inefficient, back-door workaround for a commonly needed capability in embedded real-time applications.

Mode changes require the priority of a task to change dynamically.

Examples of such mode changes include mission mode changes or fault recovery situations such as the following:

> Example 1:
>
> > A navigation function becomes more critical at Terrain Following/Terrain Avoidance (TFTA) than when flying at 40,000 feet.
>
> Example 2:
>
> > Under normal operation, Operational Flight Programs (OFPs) are sized and timed to allow complete execution of its processes within system memory and processing constraints,

regardless of software priority. However, system faults may reduce available resources below minimum threshold where the system can no longer support execution of all software. The operating system (including Ada run-time) must understand which software processes have the greatest need to remain loaded and executing.

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

POINTERS TO STATIC OBJECTS

DATE: April 4, 1989

NAME: Toomas Kaer (Endorsed by Ada in Sweden)

ADDRESS: Ericsson Radar Electronics AB
S-431 84 Molndal
Sweden

TELEPHONE: +46 31-671543
+46 31-276043
E-mail: Kaer@MOERE3.ERICSSON.SE
Toomas@EUROKOM.IE

ANSI/MIL-STD-1815A-REFERENCE:

PROBLEM:

The following Ada Language issue has been discussed by Ada Sweden Ada 9X working group. Since there is an ALIWG LI (LI 49,50) on this issue the text below is an extract from them. AIS agrees on the problem as such but not necessarily the solution to the problem. The viewpoint of AIS is to highlight the requirement not to solve the problem.

...
LI49

Provide access values pointing to static objects - The use of pointers in Ada is so restrictive that users are forced into choosing alternative designs, which are often inferior, to compensate for the lack of adequate pointer types. This language issue would allow pointers to all data objects, including static objects.

JUSTIFICATION:

a. The use of pointers (access types) are limited in Ada to dynamically declared objects. This approach is too restrictive for some applications where the use of dynamically declared objects must be prohibited. In such an application, it is still desirable to have the ability to use pointers to statically declared objects. Examples of such applications include large data structures such a maps residing in Read Only Memory (ROM).

b. Opponents to such a feature state that this would degrade the modern software engineering aspects of the Ada language. One suggested restriction that might limit abuse of such a feature is to restrict such added objects to only those elaborated at or before the elaboration of the declarative portion of the package body. This would still provide the needed capability but lessen the probability of out of scope reference, collections deallocated out of scope, etc. This would prohibit pointers to objects declared within subprograms. (In question are objects elaborated within a generic package and/or objects elaborated within a task.)

...
LI50

c. Provide conversion between address attribute and access value - The JIAWG that there is a requirement to be able to go back and forth between the address attribute of an object and the access value that points to that object. It is unsatisfactory to have unchecked conversion be the only mechanism for this function because there is no guarantee that a compiler will implement access types in a way that supports it. This issue is very similar to LI49.

In order to do this, we have to assume that 'address and an access pointer are the same kind of things or can be converted between each other. This is not true for all compiler implementations. You can take the address of an object, convert it to an access type, but that only assesses the object, you do not know how to get to the dope vector for that object. One implementation on an 8086 architecture has access types as a single segment item, but address is a multiple segment item. Also the compiler checks access values to verify that it is in the heap. The semantics of an access type and an address can be very different. This language issue could result in a major impact to compiler vendors.

Discussion centered on the requirement for this issue. Why does one have to go between address and access? One would expect that a user would do everything with addresses or everything with pointers. Also, if LI49 is approved is this issue still needed?

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

COMPILEDATE

DATE: September 18, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

It is not possible within an Ada program to discover the compilation date and time of a compilation unit.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

In compilation units requiring this, enter a dummy declaration:

```
compilation_date_time : constant string := "9999999999999999";
```

Then write a special preprocessor which searches for this string and edits it before each compilation with the appropriate information.

POSSIBLE SOLUTIONS:

1. Add an attribute "compilation_date_time" to the language which is applicable to any compilation unit and which will return the compilation date and time as a string of 16 characters in the format "yyyymmddHHMMSSFF" for year, month, day, hour, minute, second, centisecond.

   Example of usage:

   ```
   version : constant string := main_program'compilation_date_time;
   ```

   would create a constant string equivalent to

   ```
   version : constant string := "1989091812463002";
   ```

   if compiled on 1989 Sept 18 at 12:46:30.02. Compilers running under operating systems not capable of such accuracy should round to the nearest possible time unit. The numeric string format was very specifically chosen for simplicity. If the user wants other formats, slices may be taken and conversion done with the integer'value function.

Example: Assuming Month_names is appropriately declared and visible, To get the month as a word, use:

```
Month_names( integer'value( version( 5..6 )))
```

Other formats may be computed WITH package CALENDAR.

2. Add an attribute "compilation_date_time" which returns something of type CALENDAR.TIME; this is less desirable, since it requires the use of package CALENDAR for all such instances, and attributes should be package independent wherever possible.

3. Define type TIME in package STANDARD and delete it from CALENDAR in order to make option 2 more independent of compilations. If type renaming is implemented, they make CALENDAR.TIME rename STANDARD.TIME.

The author recommends option 1 as being satisfactory but which interferes with the existing language minimally.

HETEROGENOUS PROCESSING

DATE: September 21, 1989

NAME: Jeremy James (Canadian AWG #008)

ADDRESS: DY-4 Systems
21 Credit Union Way
Nepean, Ontario Canada
K2H 9G1

TELEPHONE: (613) 596-9911

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Systems that use multiprocessors interconnected with shared memory will consistently work only when homogenous microprocessors are used. An Ada program is not sensitive to the ordering of data even though the LRM implies that it is transparent to the underlying hardware. There is no mechanism for example in package system or a pragma that allows the developer to define the ordering. In addition byte ordering within a word and word ordering within larger structures are not uniform and not specifiable.

SUPPORTING POINTS:

1. Heterogenous multiprocessor that combine big endian with little endian processors cannot presently communicate unless a special protocol is defined. Such protocols at the application level can degrade performance sufficiently to eliminate some reasons for using a multiprocessor.

2. New microprocessors can support either data orientation. How is the user to define which shall be used.

DYNAMIC BINDING

DATE: September 21, 1989

NAME: Jeremy James (Canadian AWG #009)

ADDRESS: DY-4 Systems Inc.
21 Credit Union Way
Nepean, Ontario Canada
K2H 9G1

TELEPHONE: (613) 596-0574

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

The LRM specifies that binding is statically defined at run time. Systems that adapt their behavior to external events do not fit this type of binding model. Examples are: command and control, process control, call processing, fault tolerant systems, data base management systems, resource managers.

Further this severely limits the usefulness and efficiency of generic packages, because you cannot have a generic package that instantiates another generic package at run time without severe overhead.

SUPPORTING POINTS:

1. The design of severs becomes inefficient and overly complex. Severs are arbitrarily connected to at run time because a real time system is asynchronous and all processes may or may not be required to connect a predefined times. This requires the use of entry families to solve. Entry families are inefficient, difficult to understand, hard to debug and hard to maintain.

2. A software update will require that the entire system be taken down.

3. There is no mechanism to support a loadable device driver.

4. Interfaces to stream or pipe oriented communication modes cannot be developed generally, which limits run time reconfigurability. That is only predefined configurations can be supported.

SCHEDULING ALGORITHMS

DATE: August 28, 1989

NAME: Jeremy James (Canadian AMG 015)

ADDRESS: DY4 Systems
21 Credit Union Way
Nepean, Ontario Canada K2H 9G1

TELEPHONE: (613) 596-9911

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

To see if some user selectable scheduling algorithms are required rather than either

a. Using what is available in the Run-Time System, or

b. Modifying the Run-Time System to suit the particular application.

As embedded systems typically have time constraints for performing some, or all, processing, the scheduling algorithm used in the Run-Time Executive is very important. We would like to have a choice of scheduling algorithm to be available to fit the type of embedded system. We would also like the scheduling in the RTE to be more visible and controllable.

SUPPORTING POINTS:

1. One scheduling algorithm will not work well for all types of embedded system. We identified three basic types of real-time embedded system:

   a. Systems with regularly occurring events.
   b. Systems with randomly occurring events.
   c. Systems which are a mixture of the above.

2. Having a choice will give the user explicit control over the scheduling algorithm being used so that it will better fit the type of system being developed. It may also, hopefully, lead to standard approaches being used by compiler vendors, even to a standard RTE interface. At the least, the scheduling used will become visible.

3. A choice of standard scheduling algorithms, with the same basic choice being available for all Run-Time Executives, will make the scheduling visible and controllable for the user.

4. Some systems need hard deadline scheduling. If hard deadline scheduling is one of the options then users will not need to amend the RTE supplied by the compiler vendor.

5. Scheduling must be deterministic. This is particularly important in an embedded system. One of the criticisms of Ada has been the lack of determinism in the Ada tasking model.

INABILITY TO RENAME/APPEND DATA TO EXISTING FILE

DATE: September 13, 1989

NAME: Jeff Loh

ADDRESS: Intelligent Choice, Inc.
2200 Pacific Coast Highway, Suite 201
Hermosa Beach, California 90254

TELEPHONE: (213) 376-0993

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Inability to rename a file or to append data to the existing file.

All the IO packages seems to have left out this operations (except Direct_IO on append). Any substitute will not be portable.

```
procedure Rename(File : File_Type, NewName : string);
```

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Use of non portable and compiler dependent routines or system calls (also implementation dependent).

POSSIBLE SOLUTIONS:

Add the subprograms to all the appropriate IO packages.

INABILITY TO DO FINALIZATION CODE IN A PACKAGE

DATE: September 13, 1989

NAME: Jeff Loh

ADDRESS: Intelligent Choice, Inc.
2200 Pacific Coast Highway, Suite 201
Hermosa Beach, California 90254

TELEPHONE: (213) 376-0993

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Inability to do finalization code in a package

Consider the following example:

```
with ScreenManager;
procedure Main is
begin
 DoSomeThing;
end Main;
```

When ScreenManager is elaborate, the initialization code will change the screen into graphics mode. On completion, the application has no way of reverting the screen to text mode. A subprogram must be defined in ScreenManager and called by the application program to accomplish this. This violates the principle of information hiding.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Define a subprogram to do the finalization code in the application program. This violates the principle of information hiding and defensive programming.

POSSIBLE SOLUTIONS:

Taking a similar approach from UCSD Pascal version 4, where finalization code is possible,

```
package body P is

        ...

begin
 DoInitializationCode;
```

```
when package terminate => DoFinalizationCode;   -- (1)

exception
        when SOME_ERROR => DoSomething;

end P;
```

NOTES:

a. (l) must be the last statement.
b. It is not reachable by a goto statement.
c. Executes if and only if when the main program exits.

VARYING LENGTH STRING TYPE

DATE: September 27, 1989

NAME: K. Buehrer

ADDRESS: ESI
Contraves AG
8052 Auerich, Switzerland

TELEPHONE: (011 41) 1 306 33 17

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

It has been noticed, that the predefined type standard.string is unsuitable to represent strings which vary in length during their lifetime. A predefined type to represent varying length strings is thus required. This type should have the following characteristics:

```
TYPE varying_string (max_length : natural) IS
   RECORD
         length  : natural := 0;
         value   : string (1.. max_length);
   END RECORD;
```

All the operations of type standard.string are needed for varying strings as well. Assignment and comparison between two varying string objects must be possible and deliver reasonable results, even if their maximal length is different. Moreover, compatibility between varying strings and standard.string expressions would be nice.

IMPORTANCE: IMPORTANT

Varying length string types have been implemented many times, at many places, by many different Ada programmers, in many different ways. All those implementations are of course incompatible, more or less implementation-dependent and non-portable. There is no reason why this unpleasant situation should in the future, unless a varying length string is predefined by the language.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The solutions with least impact on existing implementations is to add another name to the list of predefined library units required by the standard. This library unit, say "PACKAGE strings", will implement the varying string type and all it operations. However, the operations assignment and equality cannot be simply defined, at the time (see request for overloading of "=" and ":=").

AUTOMATIC GARBAGE COLLECTION

DATE: August 4, 1989

NAME: Captain Eric N. Hanson, PhD

ADDRESS: Artificial Intelligence Technology Office
WRDC/TXI
WPAFB, OH 45433

TELEPHONE: (513) 255-1491
AV 785-1491

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Lack of automatic garbage collection

IMPORTANCE: IMPORTANT

Would help reduce the difficulty of building software and eliminating many bugs (storage leaks).

CURRENT WORKAROUNDS:

Manage storage explicitly in user code.

POSSIBLE SOLUTIONS: NONE

EXTENDING ADA TO BE TRULY OBJECT-ORIENTED (I)

DATE: August 4, 1989

NAME: Captain Eric N. Hanson, PhD

ADDRESS: Artificial Intelligence Technology Office
WRDC/TXI
WPAFB, OH 45433

TELEPHONE: (513) 255-1491
AV 785-1491

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Need to extend the Ada language to allow for inheritance of both instance variables and procedures (methods) as in Smalltalk and C++.

IMPORTANCE: ESSENTIAL

Inheritance greatly facilitates code re-use and allows more compact expression of ideas.

CURRENT WORKAROUNDS:

Program in a conventional non-object oriented style.

POSSIBLE SOLUTIONS:

Allow inheritance of methods and data from supertypes in a type (or package) hierarchy.

EXTENDING ADA TO BE TRULY OBJECT-ORIENTED (II)

DATE: August 4, 1989

NAME: Captain Eric N. Hanson, PhD

ADDRESS: Artificial Intelligence Technology Office
WRDC/TXI
WPAFB, OH 45433

TELEPHONE: (513) 255-1491
AV 785-1491

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Need to extend the Ada language to allow for polymorphism, as in Smalltalk and C++, whereby different objects can respond differently to the same message.

IMPORTANCE: ESSENTIAL

Polymorphism greatly facilitates code re-use and allows compact expression of ideas. It also eliminates many IF and CASE statements.

CURRENT WORKAROUNDS:

Use packages, which allow a restricted from of polymorphism without inheritance.

POSSIBLE SOLUTIONS: NONE

EXTENDING ADA TO BE TRULY OBJECT-ORIENTED (III)

DATE: August 4, 1989

NAME: Captain Eric N. Hanson, PhD

ADDRESS: Artificial Intelligence Technology Office
WRDC/TXI
WPAFB, OH 45433

TELEPHONE: (513) 255-1491
AV 785-1491

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

The ability to define a type hierarchy similar to Smalltalk and C++. This would involve something like allowing a package to be a sub-type of another package.

IMPORTANCE: ESSENTIAL

A type hierarchy must be defined to allow object-oriented programming.

CURRENT WORKAROUNDS:

The programmer must write code in a style that is not completely object oriented. This results in code that is more verbose and less understandable than true object-oriented style allows.

POSSIBLE SOLUTIONS:

Generalize the package construct to allow one package to be a sub-type of another package.

SIGN ATTRIBUTE FOR NUMERIC TYPES

DATE: September 26, 1989

NAME: Bryce M. Bardin

ADDRESS: E-mail: Bardin@ajpo.sei.cmu.edu

Office: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

It is potentially very inefficient to obtain the sign of a numeric type directly in Ada.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use a generic function, e.g.:

```
generic
        type Int is range <>;
function Sign (Item : in Int) return Int;
pragma Inline (Sign);

function Sign (Item : in Int) return Int is
begin
        if I > 0 then
                return 1;
        elsif I < 0 then
                return -1;
        else
                return 0;
        end if;
end Sign;
```

This function must be tediously instantiated for each integer type for which the sign is desired. As very few implementations share generic bodies, a large amount of redundant code is generated if this function needs to be instantiated for many numeric types.

POSSIBLE SOLUTIONS:

Add an attribute for all numeric types:

P'Sign For a prefix P that denotes an object of a numeric type:

If the type of P is an integer type, yields 1 if P is positive, 0 if it is zero, and -1 if it is negative.

If the type of P is a real type, yields 1.0 if P is positive, 0.0 if it is zero, and -1.0 if it is negative.

The value of this attribute has the same type as P'Base.

ALLOW DEFAULT INITIALIZATION

DATE: September 26, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

It is often desirable for reasons associated with the correctness of software to initialize objects. Many times the initial value is an attribute of the entire class of objects, i.e., of the type itself, so that providing a default initial value for all objects of that type which are declared or allocated is the obvious way to achieve the goal of initializing all such objects for correctness. It is methodologically unsound to distribute the initialization to the point of declaration of the object in such cases.

The most common situation is that a scalar type or an array of scalars or an access type should have a default value, but the point of declaration of the object is far removed from the point of declaration of the type and it is easy to forget to initialize the object appropriately.

Meaningful (application specific) default initialization cannot currently be provided for anything other than record types, but it should not be necessary to force the use of a record type solely to achieve the additional safety of centralized initialization. When a value of a scalar type is assigned to a variable, the subtype of the value is checked against the subtype of the variable. When a record type is used to enable default initialization of a scalar type, it is the subtype of the record type which is checked against the subtype of the variable, not the subtype of the scalar component, so that there is also an offsetting loss in type safety in assignment.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Initialize each object individually; or make most types open record types, private types implemented as record types, or access types designating record types.

However, most applications will not be willing to pay the additional costs for design, coding, and executing code using record types simply to avoid the chance of missing an initialization.

Without direct support for type initialization, programmers will continue to write less robust software.

POSSIBLE SOLUTIONS:

Allow the specification of default values for any type, including scalar types, array types, and private types (implemented as any type, not just as record types). This would be an upward compatible change which should be relatively inexpensive for implementations since much of the required functionality already exists in the required support for initialization of record types.

For example:

```
package X is

        type I is range 1 .. 2 := 1;

        type E is (E1, E2) := E2;

        type A is array (E) of I := (E1 => 1, E2 => 2);

        type AE is access E := new E'(E1);

        type P is private;

private

        type P is new Integer := 0;

end X;
```

CONTROL OVER VISIBILITY OF LIBRARY UNITS

DATE: September 25, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Ada supports the principle of information hiding and encapsulation through a number of mechanisms, such as user-defined types, private types, the separation of specifications from bodies for subprograms and packages, but provides no way of hiding the internal structure of a group of cooperating library units from potential abuse.

It is common practice, because of the difficulty of re-exporting entities in Ada, and frequently desirable for reasons of modularity, reusability, and maintainability, to construct abstract subsystems out of multiple library units. However, there is no way of preventing unwanted access to interfaces within the group of library units which, by design, should be private to that group. This leads to less reliability and robustness of such subsystems and to greater potential for subversion of the original intent of the design during maintenance.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

1. Provide a meta-structure for library units, which defines the relationships between groups (subsystems) of library units by delimiting their interfaces with library units not within the group. Such a structure could also support multiple versions of subsystems (e.g., different system builds for different development phases or for different targets).

2. Alternatively, at least provide a mechanism (e.g., an import list), which defines which group of units are allowed to gain visibility of ("with") a given unit (containing the list), thereby hiding it from units not in that group (i.e., not mentioned in the list).

WEAKLY-TYPED CALLS

DATE: September 25, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Applications using closely related strong types extensively (see section 7.3 of the Rationale for a discussion of "simple strong typing") in order to enhance the correctness and robustness of the code find that in many cases they must incur significant coding effort and run-time size and/or speed penalties to do so.

In particular, calling interfaces to weakly typed code (such as database management systems, graphic subsystems, communications systems, etc.) require either explicit type conversions on arguments, many instantiations of generic units (with the likelihood of an explosive expansion of code in most existing implementations), or many subprogram interfaces using pragma Interface to escape to other languages. In 50,000 line projects there may be literally hundreds of such interfaces and thousands of calls to those interfaces. The effort and costs seem to many real projects to be needlessly out of proportion to the benefits to be gained by the strong typing, so they adopt a less robust approach using subtypes or *even* predefined types. Another undesirable technique that is sometimes adopted is to use formal parameters of type System.Address and pass the addresses of objects rather than their names as the actual parameters.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Some projects for some customers are willing to pay for both the additional effort and run-time costs in using strong typing fully, but clearly would rather see a less costly approach which achieved the same level of type safety. The important point is that the typing *must* be weakened at these interfaces, only the method of achieving the weakening is at issue. The premise here is that the weakening should be minimized, but also not too burdensome to the programmer, so that programmers will not be tempted to use less desirable approaches.

POSSIBLE SOLUTIONS:

Allow programmers to use a single declaration of the desired weakening of the type model at the point of declaration of the weakly typed subprogram instead of tediously re-iterating the currently required

incantations (for explicit type conversions, generic instantiations, or pragma Interface) either at the point of call for each permutation of the set of types they need or at the point of declaration of each type (when that suffices) or at the point of declaration of each strongly-typed subprogram which interfaces to the underlying weakly-typed subprogram.

A pragma could be provided that weakens the strong typing model only slightly, from name equivalence to a form of structural equivalence. The requirement for matching of scalar actual parameters to formal parameters would be something along the line that the internal representations of the types must conform, for instance:

1. discrete actual parameters match discrete parameters as long as they have the same base type and the range of the actual position numbers is not wider than the range of the formal position numbers;

2. fixed point actual parameters match fixed point formal parameters as long as they have the same base type and the range of the actual is not wider than the range of the formal;

3. floating point actual parameters match floating point formal parameters as long as both representations have the same base type and the range of the actual is not wider than the range of the formal;

4. actual parameters of access or private types match formal parameters of access types or private types unconditionally (beware!); and

5. rules 1 through five would apply transitively to scalar subcomponents of composite types.

Appropriate rules would also be needed for conformance of record types and array types. An idea here is to allow rules similar to those for matching generic formal private types, if feasible.

PROVIDE IMPROVED SUPPORT FOR INTEROPERABILITY

DATE: September 25, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

This request attempts to provide an integrated view of the requirements which may be imposed on the Ada language and its implementations by the need for interoperable programs. It is intended to help focus arguments and stimulate discussion regarding the necessary interactions of features and interpretations required to support interoperability. It summarizes the need for capabilities that currently affect the deliberations of the Ada Rapporteur Group (ARG) and the Uniformity Rapporteur Group (URG), but which also ought to be considered in the Ada 9X revision process.

Interoperability is defined here as the capability of a program compiled with different versions of a given compiler or with different compilers to produce effectively identical outputs, given identical inputs, barring timing-related issues and provided that the hardware is sufficiently similar. This property is important for long-lived systems which undergo many changes, particularly when the system is distributed and contains heterogeneous computers.

In applications requiring interoperability, precise control over the representation of data passing through the program's external interfaces (size and meaning of bits) must be possible (within limits imposed by machine architectures) in order to match formats dictated by external considerations. Various applications have differing needs with regard to the fineness of control over representations and, for instance, whether unsigned or biased representations will be needed for discrete types.

Currently, Ada does not fully support interoperability. It is deficient in at least the following areas:

1. the effect of representation clauses is not fully specified (currently being addressed by the ARG) - unsigned and biased representations are not addressed by the standard

2. the effect of pragma Pack on arrays and records is not fully specified (also being addressed by the ARG)

3. the (minimal) required effect of Unchecked_Conversion is not well-defined

4. too much freedom is allowed in ordering of bits and in the extent of bit-numbering in record representation clauses

5. the permissible optimization which remove redundant constraint checking are not well-defined and may be implemented in such a fashion as to prevent straightforward validation of data input to the program from external files or devices, data returned as out or in out parameters from subprograms implemented through pragma Interface, or data computed with checks suppressed.

Generally, data on the interfaces of a program is first seen by Ada simply as a bit pattern in a (weakly-typed) buffer. Thus, it must be possible to map strongly-typed data in externally- defined formats from and to arbitrary portions of packed bit- and byte-arrays in order to input and output heterogeneous data as formatted messages. (Note that Ada's variant record types are not sufficiently general to construct many data-driven message formats.) After weakly-typed input data is transformed into strongly-typed data, it is frequently desirable to further convert it into the compiler's default representation (using either explicit conversion between derived types or explicit conversion between numeric types) for reasons of efficient access.

Inverse transformations may also be needed, of course. As a corollary of the fact that it is only the data passing through the program's external interfaces which is of interest, the representation of an object when it exists in an internal machine register and the default representations chosen by a compiler for data in memory are (largely) irrelevant for interoperability, since the representation of all data seen at the program's interfaces must be fully specified in order to guarantee interoperability.

It follows, then, that the compiler should be free to assign any size it likes to a scalar (sub)type by default. Further, it is reasonable for the default size of objects of a particular subtype to be context dependent, provided that the behavior is predictable. In the presence of pragma Optimize(Space), the size of an object of a given type may be smaller than it is in the absence of the pragma, the size of a scalar component in a packed array may be smaller than a simple object, and a scalar object of a constrained subtype may be stored in less space than an object of its base type. Finally, an object in an internal machine register may well be represented using more bits than when that same object is stored in memory. However, whenever a length clause is given for a scalar type it must be obeyed exactly (or rejected) whenever a simple object of that type is stored in memory. That same length clause must apply with equal force to all subtypes of that type. That is, the implementation is not free to store simple objects of a subtype in smaller space than objects of the type (or first-named subtype).

Similarly, the default layout of composite types should be up to the compiler, but record representation clauses must be obeyed exactly and must be able to exclude hidden components from the specified representation. (It is not clear to me what should be done about hidden components in arrays of such records when pragma Pack and/or length clauses apply.) Pragma Pack applied to an array type must result in no gaps between components. The compiler may assign less space for scalar components of a packed array than a simple object would occupy, but only in the absence of a length clause for the component type. In addition, it should be possible to achieve a densely-packed, unconstrained array type of 1-bit predefined Booleans by a uniformly-implemented method (i.e., by applying pragma Pack), even if the default size of type Boolean is not 1 bit. (It is not clear that this extreme form of packing should apply to other types of components, however.) (It should be noted here that the uniform availability of unsigned types with based numeric literals and logical operations would largely obviate the need for fiercely packed Boolean arrays in many applications.) Pragma Pack applied to a record type is harmful with respect to interoperability because its effect is not well-defined; it must not be used when interoperability is required.

Finally, it must be possible to perform unchecked conversions between any two objects, provided only that they have the same (logical) size. If a scalar type has an applicable length clause which forces either an unsigned or a biased representation, then the bit pattern retrieved in unchecked conversion must be that unsigned or biased pattern, not the compiler's default representation. (There is at least one existing

implementation which unbiases the value before conversion!) Array descriptors must not be considered to be part of the bit pattern representing an array object. It must be possible to define record representations which exclude implementation-dependent components from the (logical) representation of non-variant record types, at least. (The implementation is free to use any part of the space reserved by a length clause which is not accounted for by component clauses in any way that it wishes, consistent with Ada semantics.)

One possible consistent model is that the "logical" representation of an object (and its associated size) is the representation (and corresponding size) of a canonical, simple object of the type (with or without a length clause, but not as a component of a composite type) as stored in memory. Any additional information the compiler needs to support the implementation view of the data must be invisible to the user, i.e., not considered to be part of the logical representation. It then becomes the compiler's job to determine from context when and if conversions to and from the canonical form are required. Unchecked conversions of scalar components of composite objects must be performed on the canonical representation and, therefore, the canonical size. This means that implicit transformations will be needed in order to extract the appropriate bit patterns when the component is not represented in the canonical form. On the other hand, unchecked conversions of entire composite objects must preserve their specified representation when a record representation clause or pragma Pack or a length clause applies.

Two additional vexing problems which occur in attempting to write interoperable code are caused by the freedom that Ada allows with regard to record representation clauses. Both the order in which bits are numbered and whether to allow bit numbering to extend across storage boundaries are implementation-dependent. There is no technical reason why a uniform approach could not be required. Maintaining multiple versions of code just because different machine architecture manuals use opposite bit numbering should not be forced upon users because of its cost and the likelihood of inconsistencies. Thus implementations should support bit-numbering across storage boundaries and a compiler mode should be provided in which the bit order can be specified uniformly without the necessity of altering any source code. Specifically, my recommendation is that they all should support a "little-endian" view of bit numbering by default, so that bit N represents 2**N when interpreted as an integer value.

Another requirement, particularly for applications which receive data from outside the program, is for a means of validating data. There are several ways in which invalid or unchecked (and, therefore, possibly erroneous) data can enter the strong typing model in an Ada program:

1. use of an undefined value
   - use of an uninitialized variable.
   - use of a scalar out parameter which was not updated by the call; likewise for a scalar subcomponent of an out parameter

2. use of Direct_IO or Sequential_IO (or similar implementation-dependent I/O packages) to read a corrupted file or a file not of the expected type
   - an implementation is permitted to omit checks "if performing the check is too complex" (and existing implementations do so)

3. use of invalid values computed and assigned (erroneously) while checks are suppressed

4. use of non-Ada code through pragma Interface
   - only subtype checks are performed on out and in out parameters (and thus, for composite types, the subtypes of subcomponents are not checked)

5. access to a de-allocated object
   - the "dangling pointer" problem

6. unsynchronized assignment to a shared variable

7. use of an overlay violating the properties of the result type

8. use of an unchecked conversion violating the properties of the result type
   - "it is the *programmer's responsibility* to ensure that these conversions maintain the properties that are guaranteed by the language for objects of the target type" (emphasis added)

9. hardware faults

Membership tests are the language feature which has the greatest potential to validate data. (Keep in mind that membership tests on objects or values of composite types only check the subtype of the object or value itself, not the subtypes of their subcomponents, so that it takes individual membership tests on all scalar subcomponents to completely test a composite type.)

However there are several problems with using membership tests because there are scalar types for which membership tests are not adequate to assure validity and which can even produce erratic and potentially disastrous results when applied to invalid bit patterns. In particular, the usual (and presumably intended) implementations of membership tests on floating point types just use comparison against the bounds of the subtype. These tests may well fail to detect unnormalized or otherwise corrupted bit patterns. Enumeration types with representation clauses which specify non-contiguous values are particularly problematic when they are used in case or if statements and unexpected bit patterns are present because compilers usually generate code that is not defensive against invalid patterns; rather it is optimized for the correct patterns, so it may do strange things with bad values. More importantly, some existing optimizing implementations take the position (with considerable support from the words of the reference manual) that no invalid values can ever be present in an Ada program, so that if a logically required check would not really be needed in order to enforce the strong typing model for valid data, they consider the check to be redundant and omit it.

It is generally infeasible to check data for validity before it is input and it would be exceedingly inefficient to do so in terms of a raw, unconverted (bit- or byte-array) format. The only currently viable alternative is to convert it (unchecked) to a specially defined type in which all bit patterns are valid, then test it against valid values of the desired final type which are also converted to the special type. This technique is tedious and error-prone (for instance, you must be very careful that these special membership tests cannot be considered redundant too!).

Otherwise, some means must be made available to insure that explicitly written membership tests following read operations, calls, or unchecked conversions which apply to the data objects involved in those operations (or to their subcomponents) are not optimized away. Similarly, it must be possible to validate single objects and objects of types to which pragma Suppress applies by writing explicit membership tests for them or their subcomponents.

In addition, membership tests on enumeration types with non-contiguous representations must be required to be implemented to correctly account for the missing values (several current implementations do not). However, membership tests on floating point types which test for all valid bit patterns would seem to be impractical, in general.

Either a general rule should be formulated that explicit checks may not be optimized away (at least under appropriately restricted conditions) or, failing that, a language-defined (required) pragma should be provided,

although a pragma would impose a greater burden on users.

The requirements for achieving interoperability which apply to Ada implementations may be summarized as follows:

1. a length clause given for a scalar type must be obeyed exactly (or rejected for AI-325 supportable reasons) and must apply equally to all subtypes of that type so that the logical size of objects of the type (the size of the type) is determined,

2. record representation clauses must be obeyed exactly (or rejected for AI-325 supportable reasons) and must be able to exclude hidden components from simple objects of the record type,

3. pragma Pack for arrays must result in no gaps between components whose size is commensurate with the hardware architecture and for arrays of Booleans must force a 1-bit representation on its components,

4. the combination of pragma Pack and a length clause for its component type must completely determine the top-level layout of unconstrained array types,

5. length clauses applied to constrained array (sub)types must not include descriptors in the size,

6. at least signed and unsigned representations of discrete types must be supported for scalar objects and for scalar components of arrays and records,

7. unchecked conversion must be possible between the canonical representations of any two objects with the same logical size (as determined by length clauses, but excluding descriptors and hidden components),

8. explicitly written membership tests on values must never be optimized away.

It is important to the success of Ada in applications needing interoperability to refine these requirements and to address them in a timely manner. In the short term, some of these requirements may be partially fulfilled through a judicious mixture of ARG and URG decisions. In the longer term, they should be carefully considered in the Ada 9X revision process

IMPORTANCE: ESSENTIAL

Most real-time applications involve many weakly-typed interfaces between Ada programs and the external world and therefore ought to be concerned with interoperability. Not supporting the interoperability of data better than is now possible in Ada will have a strong negative impact on the benefits that Ada can bring to software development.

CURRENT WORKAROUNDS: See the discussion above.

POSSIBLE SOLUTIONS: See the discussion above.

PROVIDE A STANDARD PACKAGE FOR SEMAPHORES

DATE: September 26, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Ada provides no standard way to get an efficiently implemented primitive for mutual exclusion. The Ada task construct is general. Many applications cannot afford the overhead of the general rendezvous just to achieve mutual exclusion on the large number of shared data structures they require. Achieving program correctness through the use of synchronization at a high level between a smaller number of tasks is difficult at best from the design point of view, and will often require no fewer rendezvous than synchronization at a lower level. Mutual exclusion can be implemented by encapsulating tasks with either individual data objects or with the operations on the data type, but this is frequently also too costly because every operation will incur the overhead of a general rendezvous. What is needed is a compromise that allows a coarser-grained mutual exclusion which is invocable for arbitrary sequences of actions (e.g., database transactions). Semaphores are well-suited for this purpose, but the frequency with which they must be accessed ordinarily precludes the use of a user-defined Ada semaphore. Although some implementations already provide a standard interface to efficiently implemented semaphores, which addresses this need, its usefulness is such that ought to be a standard part of the Ada language and required for all implementations as are package Calendar and Text_IO.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Less efficient semaphores may be defined by any user.

POSSIBLE SOLUTIONS:

Provide a standard specification which is required of all implementations.

A 'SPACING ATTRIBUTE

DATE: October 20, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

One problem currently still under debate in the ARG is the meaning of 'Size as applied to types and objects. It seems that there ought to be a distinction drawn between the logical size of a type (the significant, or meaningful bits of its representation) and the space allocated by the compiler for objects in various contexts.

It seems that new attributes might help to maintain the semantic distinction.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

The following is just a point of departure:

P'Spacing — For a prefix P that denotes any type or subtype:

Yields the number of contiguous bits of storage between consecutively allocated single objects of the type, including any padding or fill bits or any sign extension bits or bits reserved for alignment or storage management which the compiler maintains at the place of allocation for its own convenience of efficient access and allocation or deallocation.

If, for any reason, the spacing is not constant, this attribute yields the conventional value zero.

The value of this attribute is of the type universal_integer.

P'Spacing — For a prefix P that denotes an object:

Yields the number of contiguous bits of storage occupied by the object including any padding or fill bits or any sign extension bits or bits reserved for alignment or storage management which the compiler maintains at the place of allocation for its own convenience of efficient access or allocation and deallocation.

For objects of access types, P'Spacing is the incremental storage cost for the allocation of each (additional) designated object.

If, for any reason, the spacing is not constant, this attribute yields the conventional value zero.

The value of this attribute is of the type universal_integer.

P'Allocation For a prefix P that denotes an access type or subtype or that denotes an object of an access type: Yields the incremental storage cost for each additional allocation of a designated object.

If, for any reason, the spacing is not constant, this attribute yields the conventional value zero.

The value of this attribute is of the type universal_integer.

It is intended that the values of these attributes be computed as if the object or type was (to be) stored in memory even when, due to optimization, the object only exists in a machine register. The basic aim of this approach is to allow users to calculate memory utilization in a predictable way.

DESTRUCTOR

DATE: October 20, 1989

NAME: Wesley F. Mackey

ADDRESS: School of Computer Science
Florida International University
University Park
Miami, FL 33199

TELEPHONE: (305) 554-2012
E-mail: MackeyW@servax.bitnet

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Ada provides no way of automatically having storage which is indirectly referenced by an object automatically reclaimed when that object is deleted. For example, suppose we have a declaration in a package

```
type List is limited private;
```

with associated operations exported from that package, and then we have a client which declares:

```
a, b, c: List;
```

When a, b, and c reach the end of their scope (assume they are declared inside of a procedure), they are automatically removed, but not the storage they point at.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Define a procedure DISPOSE, which can be called, as in:

```
DISPOSE(a); DISPOSE(b); DISPOSE(c);
```

Where DISPOSE would go through the lists pointed at by these variables and suitably reclaim storage.

PROBLEM: A user might forget to call this procedure.

POSSIBLE SOLUTIONS:

Allow a pragma to be declared:

```
pragma destructor( DISPOSE, LIST );
```

which would cause the procedure dispose to be called with an argument of type LIST whenever an object of that type was to be deleted. It would then be the responsibility of DISPOSE to reclaim the storage. This could be done implicitly at the close of the scope of existence of the variables a, b, c, above.

NEED FOR MODERN, OBJECT-ORIENTED CAPABILITIES

DATE: October 29, 1989

NAME: William Thomas Wolfe

ADDRESS: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: (803) 656-2847
E-mail : wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Ada 83 does not provide modern object-oriented capabilities such as multiple inheritance, polymorphism, and dynamic binding.

CONSEQUENCE:

Ada users are unable to take advantage of the power and flexibility of the object-oriented paradigm in a clean and maintainable manner.

IMPORTANCE:

CURRENT WORKAROUNDS:

Use commercially available preprocessors, but this results in a separation between the programming level of abstraction (the object-oriented preprocessed code) and the debugging level of abstraction (the code generated by the preprocessor).

POSSIBLE SOLUTIONS:

Ben Brosgol, Vice President of Alsys, has recently written in Alsynews (a free publication of Alsys, Inc.) regarding the reasons Jean Ichbiah (Alsys Founder & CEO, and leader of the Ada 83 design team) did not provide more direct support for object-oriented programming in Ada 83:

[...] Jean Ichbiah, the principal designer of Ada, worked on one of the first implementations of Simula-67. He was well aware of the implementation consequences of classes and inheritance, and it was in fact run-time efficiency arguments that dictated against inclusion of these features. To see this, consider the representation of objects at run time. If a class SQUARE inherits from a class POLYGON, any field in a POLYGON must also be in a SQUARE, and thus the representation of a SQUARE must be at least as large as a POLYGON. However, with polymorphism it is possible to assign a SQUARE to a POLYGON. This means either (1) truncating the SQUARE-specific fields; (2) preallocating the maximal amount of space for a POLYGON that will be required for any of its heirs; or (3) using pointers and representing objects indirectly. The first alternative loses information and is not desirable.

The second alternative requires a centralized definition of POLYGON; [...] this would defeat some of the purposes of object-oriented programming. Thus the third alternative is adopted in OOLs. Assignment means either pointer copy and object aliasing, or allocation of a copy of the source object for reference by the target. However, once the fateful step of implicit pointers is taken, the implementation cannot ignore the problems of storage reclamation. If garbage collection is used (as in Simula), efficiency and applicability to real-time applications are sacrificed. If garbage collection is not used, then the programmer is left to figure out when objects are no longer referenceable and when they can thus be explicitly deallocated. [...] Since the implicit use of pointers is so unappealing, Ada avoids the problem by not providing the features responsible. Thus Ada [83], lacking some of the run-time flexibility found in OOLs, is not an Object-Oriented Language. On the other hand, what it gains is more static checks (and hence earlier detection of errors) as well as increased efficiency. Which is what is needed to program the class of real-time applications that Ada was designed to handle.

Several problems are now apparent in this reasoning. First, it is necessary to isolate the question of multiple inheritance from the question of polymorphism. Multiple inheritance is nothing more than a definitional mechanism, and hence can be completely handled at compile time. Thus run-time efficiency arguments do not apply to this aspect of the problem. Secondly, Ada is not just for real-time users; information systems users are one example of another major group of Ada users (Ada 9X Project Report, January 1989, III.4). Thus Ada 9X must take into account the needs of both real-time and not-necessarily-real-time users. The Classic Ada product (which allows Ada software developers to use inheritance and dynamic binding in object-oriented Ada designs; available from Software Productivity Solutions, Inc. at SPS@radc-softvax.arpa) provides a product feature whereby techniques can be applied to either optimize away the need for or block the use of anything involving significant run-time expense (such as dynamic binding), while still providing the advantages of those capabilities which do not require such costs (such as inheritance). Thus, the path to these capabilities in Ada 9X has already been defined: simply provide a pragma REAL_TIME which will cause the compiler to either generate code which does not expose the user to significant run-time penalties if that is possible, or generate an error message along the lines of "Statement is inconsistent with the requirements of pragma REAL_TIME" if not. And finally, significant advances have been made in the realm of real-time garbage collection algorithms, although the problem is not yet completely solved. The provision of object-oriented capabilities in conjunction with pragma REAL_TIME allows compiler vendors to incorporate sophisticated new real-time garbage collection algorithms as soon as they are discovered, and then modify the effect of pragma REAL_TIME such that it does absolutely nothing. In this way, advances in garbage collection technology will enable real-time users to begin using the full range of object-oriented capabilities almost immediately, without obsoleting any of the code written back when the effect of pragma REAL_TIME was much more restrictive.

Hence, Ichbiah's rationale for not including classes and inheritance in Ada 83 does not apply to the current Ada community. With further advances in real-time garbage collection technology, the efficiency considerations will not apply to anyone at all. The inclusion of modern object-oriented capabilities in Ada 9X will permit non-real-time users to take advantage of them immediately, and real-time users to take advantage of some of them now and the rest of them as compiler technology permits. It is therefore appropriate that the flexibility and power of the object-oriented paradigm be included in Ada 9X, and I expect that this will indeed be the case.

LIBRARY-LEVEL RENAMES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Library-level renames are not allowed by the standard, even though there appear to be no drawbacks to them, and there are no cases in which they would be useful.

IMPORTANCE: IMPORTANT

This revision request is motivated primarily by concerns about symmetry and aesthetics, but it also yields some practical benefit to programmers because there are cases in which library-level renames would clean up naming considerably.

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS: Allow library-level renames.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

MERIT BADGES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

The Ada language itself is standardized by the efforts of the AJPO and the validation tests, and this has gone a long way toward enforcing portability. Unfortunately, this has not proven sufficient, because the very first thing compiler vendors do is to supply a set of proprietary interfaces against which they encourage programmers to write their programs: the inevitable (and somewhat ironic) result of this is non-portable programs written in a portable language.

IMPORTANCE: ESSENTIAL

If something is not done to reign in this tendency, the eventual result will be to render moot all claims about Ada's portability and reusability.

CURRENT WORKAROUNDS: Programmer discipline.

POSSIBLE SOLUTIONS:

It might be a good idea for the AJPO to consider creating a second kind of validation sticker, a sort of "merit badge". For each library of standard interfaces that is supported by a particular compiler, a merit badge for that particular library will be granted. Thus, a basic compiler might only pass the basic validation suite, but a compiler with a lot of merit badges might provide support for graphics, databases, networking, etc.

The question of who defines the standard interfaces is left for someone far more diplomatic than the author.

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

NON-CONTIGUOUS SUBSETS FOR SUBTYPES OF DISCRETE TYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Subtypes of discrete types currently must be consist of contiguous ranges of elements. Unfortunately, there are many cases in which a sub type consisting of a non-contiguous set of elements from a parent type is what is really desired.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use a set package of some kind. Unfortunately, this is inefficient and non-portable.

POSSIBLE SOLUTIONS:

Extend the syntax of the subtype specification for discrete types so that it is possible to specify non-contiguous sets of elements from the parent type (this should be applicable transitively, so that the parent type can itself be a subtype, as is currently possible with contiguous range subtypes). Several possibilities exist for the actual syntax:

```
type Workdays is (Monday, Tuesday, Wednesday, Thursday, Friday);
subtype Bad_Days is Workdays range Monday & Wednesday;
subtype Bad_Days is Workdays' (Monday, Wednesday);
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

SUBPROGRAM TYPES

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3606 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

There are certain approached to programming that are perfectly valid (and which are in fact often superior to any other method for certain applications), but which are poorly-supported by the current standard. What characterizes these approaches is that they tend to be very dynamic, with the behavior of the program changing as a consequence of its interaction with the user, and with the binding of the specific actions taken deferred as long as possible.

Consider a menuing system. One way to implement such a system is to build a data structure representing the menus and to traverse this structure during program execution. Each node of the data structure contains some number of operations that are bound to keys/mice/etc. The user, arriving at some node, presses a key/mouse and is rewarded with an action appropriate for the current node: the date structure and the program that traverses the data structure work together to form a context-sensitive execution environment constituting the menuing system.

This is a very nice way to implement such a system, and many hypertext applications are moving toward this sort of implementation; this sort of implementation also has the advantage that it is easy to reconfigure, even to the point that in some implementations the user can bind his/her own operations to nodes in the structure.[1]

Now, ask yourself how you would construct such an implementation in Ada: a few minutes; reflection should be sufficient for you to conclude that doing so is somewhere between cumbersome to completely impossible with the current standard.

IMPORTANCE: ESSENTIAL

---

[1]Some may criticize this kind of programming as the spiritual equivalent of self-modifying code, but it need not be that unsafe if done carefully.

This is easy to do in other languages (such as C), and critics beat up on Ada (and rightly so) for making it nearly impossible.

CURRENT WORKAROUNDS:

It is possible to store tasks in the nodes of the data structure and to rendezvous with them to perform the desired actions. This, however, is incredibly inefficient. Similarly inefficient tricks can be played with dynamic instantiation of generics.

POSSIBLE SOLUTIONS:

Elevate subprograms to the status of first class objects, similar to tasks; add subprogram types, subprogram constants, subprogram variables, subprogram parameters (subprogram passed as arguments to other subprograms), etc.

This may seem to be very complicated and to induce large perturbations in the standard, but the syntax (described below) actually mimics the existing syntax to the point that no new keywords are required, and the lifetime issues are in many ways no worse than for tasks and task types.[2]

Some may criticize these changes because they make the language more dynamic and therefore, in theory perhaps, more dangerous. But this is not necessarily true: as long as strong-typing is maintained and lifetime issues are dealt with properly, this need not be dangerous at all. In fact, these changes can give Ada programmers the best of both worlds: the dynamism of C with the safety of Ada.

As for specifics of syntax...

Procedure and function types should be added. For example:

```
type Integer_Muncher is procedure (some_Value : in Integer);
type Integer_Cruncher is function (some_Value : in Integer)
                                                    return Integer;
```

The rules for matching actual procedures and functions to procedure and function types should be similar to the rules for matching actuals to generic formals. For example:

```
procedure Count_Proc (This_Value : in Integer) is ...
-- This procedure is of type "Integer_Muncher".

function Count_Func (This_Value : in Integer) return Integer
                                    is ...
-- This function is of type "Integer_Cruncher".
```

Constants and variables of procedure and function types should be added, and it should be possible to assign procedures and functions to them (provided, of course, that the types match). For example:

---

[2]It can be argued that if tasks types are understandable and belong in the language then it must certainly be the case that subprogram types should be supported: they can't be any more complicated and they are probably much less complicated.

```
constant Counter_1 : Integer_Muncher   := Count_Proc;
variable Counter_2 : Integer_Cruncher  := Count_Func;
```

Procedure and function parameters should be added. For example:

```
function Foo (Counter : in Integer_Muncher) return Boolean is...
```

Default procedure and function parameters should be added. For example:

```
procedure Foo (Counter : in Integer_Cruncher :=Count_Func) is...
```

It should be possible to return procedures and functions vis parameters of mode OUT and IN OUT. For example:

```
procedure Foo (New_Function : out Integer_Cruncher) is
begin
        New_Function := Count_Func;
end Foo;
```

It should be possible to return procedures and functions form functions. For example:

```
function Foo (Value : in Integer) return Integer_Muncher is
begin
        if Value > 0 then
                return Count_Proc;
        else
                ...
        end if;
end Foo;
```

It should be possible to test quality of variables and/or constants of procedure and function types. For example:

```
if Counter_2 = Count_Func then...
```

It should be possible to declare procedure and function types private. For example:

```
package Some_Package is

        type Integer_Muncher is private;

private

        type Integer_Muncher is procedure (Some_Value : in Integer);

end Some_Package;
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will recompile successfully and will behave identically during execution except for possible small changes in execution speed.

WHEN/RETURN CONSTRUCT

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rational, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95054-3197

TELEPHONE: (408) 496-3706 [11am-9pm]

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

Functions are often implemented as a set of IF statements which, if the condition evaluates TRUE, return some result:

```
function Some_Function return Some_Type is
begin
        if Some_Condition then
                return Some_Result;
        elsif Some_Other_Condition then
                return Some_Other_Result;
        end if;
        end Some_Function;
```

These conditionals contain a lot of syntactic sugar that makes reading and/or debugging them somewhat difficult (not to mention the difficulty of typing them in correctly in the first place).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Add a WHEN/RETURN construct to the language similar in intent to the EXIT/WHEN construct for loops. Using this construct, the above code sample would be rewritten as follows:

```
function Some_Function return Some_Type is
begin
        when Some_Condition return Some_Result;
        when Some_Other_Condition return Some_Other_Result;
```

```
end Some_Function;
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will behave identically during execution except for possible small changes in execution speed.

LIBERALIZATION OF END RECORD

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 89[illegible]-5060

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Most of the constructs in Ada permit logical matching ends. For example, the end which matches a begin can have a name afterward which echoes the name which corresponds to the begin. Those cases where a name is not allowed, such as end if, mostly have no corresponding name to use. There is, however, an irregularity. Although a record type declaration can be complex and involve nesting of variants there is no way to identify its end as any more than the end of some record type declaration.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

```
type Fred is record
...
end record; -- Fred
```

POSSIBLE SOLUTIONS:

Permit an optional repetition of the record type name on the end line:

```
type Fred is record
...
end record Fred;
```

ATTRIBUTES AS FUNCTIONS

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

One of the goals of the design of the Ada programming language is extensibility. It should be possible to create types in an application which are first class types, and look like they came with the language. For example, it should be possible to make a type which is not implemented as a numeric type, while making it behave as much as possible like a numeric type.

This is not possible, because the language does not permit attributes to be defined by the application. All applications of attributes are functions or constants, and all can be construed as functions. One can define a type very much like universal_integer, and call it universal_integer, with such operators as "+" and "=", while implementing it as a pointer to an array of integers, but one cannot define universal_integer'succ.

It is also not possible to define the conversions from one type to another which would exist it the type were truly numeric.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Admit the type is second class, and define a function called universal_integer_succ. Call the type universal_integers and call the conversion function universal_integer.

POSSIBLE SOLUTIONS:

Permit functions to have apostrophes in their names if and only if the suffix of the apostrophe is one of the attribute names in the language and the prefix of the apostrophe is a locally declared type.

```
function Universal_Integer'succ
        (Left: universal_integer)
        return universal_integer is
begin
        return Left + create_universal_integer(1);
end universal_integer'succ;
```

Of course, the attributes which return type universal_integer cannot be exactly modeled by functions, but

the user can select an appropriate integer type to define a function which can be used in the appropriate manner.

As is clear from the above, this does not completely solve the problem, since we still cannot use the numeric literal for an access type (presumably exported as a private type). For this, it would be desirable to permit the following extension of overloading which permits a type name to be used as the name of a function:

```
function Universal_Integer
        (Left: integer)
        return universal_integer;
```

FINALIZATION

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: None or all.

PROBLEM:

There is an asymmetry in that objects can be initialized, but cannot be finalized. Even packages have initialization clauses. There is no means to assure that a variable or package performs certain operations when it is completed.

When an access type is exported, it is desirable to have a way of assuring that all objects are removed from memory when they go out of scope. A package might open a file in its initialization clause, and should logically close that file in its finalization, but there is no way in the package specification or body to assure that the finalization routine is in fact called.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Write a procedure to finalize the object or package, and assure by procedural methods (code audits, either manual or automatic) that it is called whenever necessary.

POSSIBLE SOLUTIONS:

Add finalization clauses to type declarations and package bodies. The syntax for packages should resemble:

```
package Something is
        ...
begin
        ... -- initialization
terminate
        ... -- finalization
exception
        ...
end something;
```

The use of "terminate" is intuitive, and its occurrence without a semicolon is unambiguous.

The syntax for type finalization syntax is not so obvious. Since initialization is represented by punctuation,

finalization should also. Conceptually, finalization is more a procedure call than a function. Perhaps something like:

```
type Thing is access designee := null => erase;

procedure Erase (Something: in out thing);
```

The procedure would always have an object of the type being declared for its parameter, and could be forbidden to have any other parameters. It would always be a forward procedure (not yet declared) since it cannot be declared before the type of its parameter.

This could also be done for tasks by allowing a sequence of statements after the terminate alternative. Of course, no accepts would be allowed in this sequence of statements:

```
select
        ...
or
        ...
        terminate;
        ... -- task finalization
else
        ...
end select;
```

COMPONENT SELECTION AS A FUNCTION

DATE: August 3, 1989

NAME: Nicholas Baker

ADDRESS: McDonnell Douglas Electronic Systems Company
5301 Bolsa Avenue 28-1
Huntington Beach, California 92647

TELEPHONE: (714) 896-5060

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

In many cases, functions are provided which give access to components of record objects whose types are private. Object package specifications often consist of long sequences of update and extract procedures and functions. In addition to permitting the hiding of other details of the type implementation, these access subprograms can be used as actual generic parameters. Unfortunately, they are verbose and congest the name space.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

The multitude of access functions can be used to give the effect of exporting the fact that a type is a record.

```
package D_Argand is
        type Complex is limited private;
function RE
            (Value: complex) return float;
function IM
            (Value: complex) return float;
procedure Set_RE
            (Value: in out complex; RE: in float);
procedure Set_IM
            (Value: in out complex; IM: in float);

private
        type Complex is record
                RE,IM: float;
                Modulus: float range 0.0..float'last;
        end record; -- complex
end d_argand;

with d_argand; use d_argand;
procedure User is
        X: complex;
```

```
begin
        set_im(x, re(x));
end user;
```

POSSIBLE SOLUTIONS:

Permit componency to be declared so that users can avoid differentiating true records from apparent records.

```
package D_Argand is
        type Complex is limited private;
    select complex.IM;
    select complex.RE;
private
        type Complex is record
                RE,IM: float;
                Modulus: float range 0.0..float'last;
        end record; -- complex
end d_argand;

with d_argand; use d_argand;
procedure User is
        X: complex;
begin
        x.im := x.re;
end user;
```

Of course, it would be superior if the existence of these apparent components did not in fact require that the type be a record type with components of those names:

```
package D_Argand is
        type Complex is limited private;
    select complex.IM;
    select complex.RE;
private
        type Complex_Entity;
        type Complex is access complex_entity;
end d_argand;

package body D_Argand is
        type Complex_Entity is record
                real, imaginary: float;
                Modulus: float range 0.0..float'last;
        end record; -- complex_entity
        select complex.IM is select complex.all.imaginary;
        select complex.RE is select complex.all.real;
end d_argand;
```

Since each of these apparent components has two sides, a read function and a write procedure, their use as generic parameters must be carefully defined, but one could allow:

```
with D_Argand; use D_Argand;
package body User is
        generic
                with procedure Set_Component
                        (Container: in out complex;
                Component:  in float);
        procedure It;
        ...
        procedure Real_It is new it (complex.RE);
        procedure Imaginary_It is new it (complex.IM);
end user;
```

ALLOW NON UNITY INCREMENTS IN LOOPS

DATE: October 30, 1989

NAME: Jon Squire (topic requested by SIGAda NUMWG)

ADDRESS: 106 Regency Circle
Linthicum, MD 21090

TELEPHONE: (301) 765-3748
E-mail: jsquire@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

A small percent of the time it is very inconvenient to convert a "for" loop to a "while" loop. Most of the inconvenience comes from not having a loop increment.

IMPORTANCE: IMPORTANT

Mostly, it seems strange to new users that they can not write

```
for I in A'RANGE step 2 loop
  ...
end loop;
```

CURRENT WORKAROUNDS:

```
I : INTEGER;

I := A'FIRST;
while I <= A'LAST loop
   ...
  I := I + 2;
end loop;
```

POSSIBLE SOLUTIONS:

"step" is not a reserved word. It is a word that is recognized in a particular syntactic structure. The literal constant 2 in the example could be any expression that was of the type of the loop variable.

SUBSETS

DATE: October 31, 1989

NAME: Gerald L. Mohnkern

ADDRESS: DARPA/ISTO
1400 Wilson Blvd
Arlington, VA 22209

TELEPHONE: (703) 522-2371
E-mail: mohnkern@nosc.mil

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Ada is the only widely used language which explicitly provides for MIMD parallelism, but it is infrequently used for distributed memory MIMD computers because its large and complex run-time package must be replicated in each processor.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Using extensions to other languages instead of Ada for MIMD computers.

POSSIBLE SOLUTIONS:

Define a standard subset of Ada. This subset should exclude those features of Ada which increase the complexity of the runtime system, without destroying its functionality. (I realize that is a pretty large order.)

ALLOW PREFERENCE CONTROL FOR ENTRIES IN A SELECT STATEMENT

DATE: October 31, 1989

NAME: P. N. Lee

ADDRESS: University of Houston - University Park
Dept. of Computer Science
4800 Calhoun Rd.
Houston, TX 77204-3475

TELEPHONE: (713) 749-3144

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

The selection of an entry from several open alternatives in a select statement is not defined by the language. However, some tasks, in particular, server tasks, may have some entries which are more important than others. For example, a server task which manages a pool of data buffers may prefer to accept a request to release a buffer over a request to acquire a buffer. The ada language does not define a simple mechanism to control this preference.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Guards involving the 'COUNT attribute. This construct, however, is complicated, and may be quite lengthy if a large number of entries are present. Also, it may cause unnecessary blocking when a call to a high preference entry is cancelled during the evaluation of the guards.

POSSIBLE SOLUTIONS:

Include a preference directive in association with the accept statement inside a select statement. For example:

```
select
        preference => 8 accept E1 ...
or
        preference => 4 accept E2 ...
end select;
```

ALLOW SCOPE OF INLINED SUBPROGRAM TO BE COMBINED WITH ENCLOSING SCOPE

DATE: October 30, 1989

NAME: J. M. Holdman, A. R. Leifeste

ADDRESS: Shell Development Company
P.O. Box 481
Houston TX, 77001

TELEPHONE: (713) 663-2260
E-mail: jon@shell.com

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

When inlining a subprogram that contains declarative items and/or an exception handler, a block must be used to surround the inlined code. If global optimization is performed by the compiler, the presence of these blocks may inhibit the optimization.

IMPORTANCE: IMPORTANT

The software engineering mindset of Ada is to write modular, cohesive subprograms. To gain efficiency, however, it may be necessary to incline some of these subprograms. A highly optimizing compiler should be able to perform some optimization across inlined subprogram bodies. If local blocks must be used to contain local declarative items and/or exception handlers, it is unlikely that global optimization could be performed.

CURRENT WORKAROUNDS:

1. Explicitly write all code inline, without using subprograms.

2. Keep subprograms very simple; that is, do not use local declarative items or exception handlers.

POSSIBLE SOLUTIONS:

Allow sufficient code motion that exception handlers of inlined subprograms could be combined with exception handlers of the enclosing scope, and that declarative regions of inlined subprograms could be combined with the declarative region of the enclosing scope. A restriction that the declarative region of the subprogram to be inlined contain only basic declarative items would be reasonable. Some type of identifier renaming may be required to avoid ambiguity and visibility problems.

ADD VECTOR SYNTAX AND SEMANTICS

DATE: October 30, 1989

NAME: W. R. Lee

ADDRESS: Shell Development Company
P.O. Box 481
Houston, TX 77001

TELEPHONE: (713) 663-2379
E-mail: lee@shell.com

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

The goal of Software Engineering is to achieve understandable, modifiable, reliable, and efficient software. It is commonly accepted that a problem solution should reflect the problem that it is solving, and this, coupled with readability of the software, it inherent in modifiable, understandable software. Efficient software utilizes the computing resources optimally.

A significant amount of computing involves high performance vector processing computers. Companies such as Cray and Convex owe their existence to the need for such computing. And, in fact, much of the reputation of our country's computing capability is in our ability to produce "the biggest" and "the fastest" supercomputers in the world.

Historically, using high performance vector computers has been the domain of the FORTRAN programmer. vendors who build such computers provide FORTRAN compilers that automatically convert standard code into efficient code for the vector processing hardware. However, code for which automatic vectorizers work best is often extremely cluttered and very poorly structured. Vectorizers need as much code embedded into the DO-loops as possible in order to operate successfully, resulting in code which is very difficult to read and understand. Typical FORTRAN code that vectorizes well bears little resemblance to the basic code that would normally be written, and such code becomes a maintenance burden.

The choice to write vectorizing compilers is driven by three primary items:

1. Huge amounts of existing FORTRAN code exists which, if recompiled, could be improved by a compiler that automatically infers the underlying vector processing that is represented by the FORTRAN syntax.

2. No language standard exists (from a historical perspective) that allow vector syntax in FORTRAN 8x efforts. This lack has been realized by the FORTRAN world, and just such syntax and semantics have been proposed.

3. Since no language syntax exists, any new extension will have to be learned by the FORTRAN programmers. ALL FORTRAN programmers can write DO-loops, but all would have to be taught a new syntax and semantics.

Items 1 and 3 do not apply to the Ada community: there is not a huge amount of "dusty deck" Ada

kicking around waiting to be vectorized, and moving to Ada requires training in a new language in any case. Only the second point needs to be addressed.

Ada can be the language of the future. It brings benefits far beyond its language definition. But to efficiently employ Ada on supercomputers, a means for allowing compilers for those computers to employ the vector functions must be found. There are two proposed mechanisms: 1) provide minor changes to the existing language definition which, coupled with additional PRAGMAs, would allow vectorizing Ada compilers to be created; and 2) define an addition to the language to explicitly allow vector syntax and semantics. Several language change proposals have already been entered for the former; this proposal is for the latter.

IMPORTANCE: ESSENTIAL

A significant community uses very-high performance super computers to do its computing work. The software engineering mindset that Ada "brings to the table" is as important to that community as elsewhere. The lack of highly efficient Ada software for computers like a Cray or a Convex inhibits deploying Ada into that community. Since a large amount of the mathematics which is ultimately computed on vector processing machines is represented in vector form, a vector syntax in the language would allow problem solutions to mirror problem statements.

CURRENT WORKAROUNDS:

1. PRAGMA INTERFACE to FORTRAN routines

2. The beginnings of vectorizing Ada compilers with "best case" projections of only 85% of the capability of FORTRAN. Using these Ada compilers results in the same unreadable, unmaintainable code that exists in today's FORTRAN systems.

POSSIBLE SOLUTIONS:

Vector types and operations must be defined. A list of basic operations which access the power of the vector machines can be obtained from any vendor. Attention must be given to the relationship of vectors to simple arrays of other types, e.g., floating point.

Inclusion of a vector syntax would allow super computer vendors to take full advantage of their systems while allowing the programmer to write more readable, understandable code. Vendors without vector capability would provide Package STANDARD support that would perform the vector operations with standard sequential code precisely equivalent to the corresponding support for the existing types.

NEED FOR IMPROVED LANGUAGE CONSTRUCTS FOR PARALLEL AND DISTRIBUTED PROGRAMMING

DATE: October 30, 1989

NAME: Dennis J. Philbin

ADDRESS: Scientific Computing Associates, Inc.
246 Church Street, Suite 307
New Haven, CT 06510

TELEPHONE: (203) 777-7442
E-mail: yale!sca!philbin

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM:

The decade since the Ada design effort has seen an explosion in the technical community's understanding of parallel and distributed programming. It's now clear that, while "Core Ada" remains a sound design, Ada tasking and IPC are in many cases too expensive and not flexible enough to serve as good bases for distributed, parallel and realtime applications. Tasking and IPC in Current Ada are poorly suited to the needs of master-worker parallelism--an approach that has emerged as one of the simplest, most robust and most efficient ways to write parallel applications. They impose synchronization and naming constraints that are excessively rigid, hence both are inefficient and hard to use. They don't support a full spectrum of grain sizes for parallel applications. They provide poor support for database applications. They are inappropriate for use in heterogeneous or evolving environments.

IMPORTANCE: ESSENTIAL

It is essential that parallel and distributed applications be supported with language constructs that are easy to use, to read, to support efficiently and to adapt to the full spectrum of hardware environments and software needs.

CURRENT WORKAROUNDS:

The problems with Ada tasking and IPC are extensive and varied. There's no standard set of current workarounds. Ada programs have tended to avoid certain desirable programming styles altogether rather than to support them using clumsy, unidiomatic and inefficient language constructs.

POSSIBLE SOLUTIONS:

Core Ada plus Linda operators (Ada Lite) will be a better vehicle than current Ada for parallel and distributed programming.

4.1 New Operators

Proposed new constructs consist of six statements that support Linda's tuple space abstraction. Tuple space is a mechanism for creating and coordinating multiple execution threads. Tuple space is a bag of tuples,

where a tuple is simply a sequence of typed fields, for example("new stuff", 0, 16 01)--a 3-tuple that consists of a string, an integer and a real. The new statements proposed provide mechanisms for dropping tuples into the bag, hauling tuples out and reading them without removing them. These mechanisms take care of all inter-process communication and coordination needs. Basically, if task R has some data for task S, R puts the data in a tuple and drops the tuple into tuple space. S can either read the tuple or haul it out, depending on circumstances. The new statements also support the creation of "live tuples", whose fields aren't evaluated until after the tuple enters tuple space. When a live tuple is added to tuple space, it is evaluated independently of, and in parallel with, the task that dropped it in. When it's done evaluating, it turns into an ordinary data tuple that can be read or removed like any other. This mechanism supports task creation: to create one hundred parallel processes or tasks, drop one hundred live tuples into tuple space. To find a particular tuple, we use associative look-up: tuples don't have addresses; to locate the one we want, we search on any combination of field-values.

There are four basic tuple-space operations, out, in, rd and eval, and two variant forms, inp and rdp. out(t) causes the tuple t to be added to TS; the executing process continues immediately. In (s) causes some tuple t that matches the template s to be withdrawn from TS; the values of the actuals in t are assigned to the formals in s, and the executing process continues. If no matching t is available when in(s) executes, the executing process suspends until one is, then proceeds as before. If many matching t's are available, one is chosen arbitrarily. Rd(s) is the same as in(s), with actuals assigned to formals as before, except that the matched tuple remains in TS. Predicate versions of in and rd,inp and rdp, attempt to locate a matching tuple and return 0 if they fail; otherwise they return 1, and perform actual-to-formal assignment as described above. Eval(t) is the same as out(t), except that t is evaluated after rather than before it enters tuple space; eval implicitly forks a new process to perform the evaluation.

Syntax, detailed semantics, and proposed integration of these operators with Ada type and object-structure system are discussed in [SCA TR 148].

4.2 Support for distributed data structures

Efficient support for distributed data structures is a crucial part of Linda's power and simplicity. The distributed data structures that often form a basis for the most natural and most efficient solution to a parallel programming problem can't be represented in current Ada. Consider the master-worker technique for parallel programming. Such programs create a collection of identical worker processes, each prepared to perform whatever task is assigned. The tasks may be intended for execution in some fixed order or in arbitrary order. Because they are handed out dynamically--rather than pre-assigning tasks to workers, we let each worker grab a fresh task whenever the previous one is complete--these programs tend to be naturally load balancing. (A parallel program can't be effective unless the work is spread uniformly among the available processes.) Current Ada militates against this powerful, simple and robust style of building parallel and distributed applications. Ada Lite strongly supports it.

Master-worker programs depend on distributed data structures--structures that are directly accessible to all processes in the program--for storing the problem description, task assignments and evolving program state. These shared synchronized-access structures (bags, queues, arrays, tables) must (in current Ada) be placed in the keeping of some dedicated manager processes. In Linda-extended Ada, they can be placed in tuple space, where all interested processes can access them directly and (up to the necessary consistency constraints) simultaneously. A manager process is centralized and a distributed data structure is (obviously) distributed. A manager process may accordingly impose performance costs, in terms of extra transactions between user and system code. It may be a performance bottleneck--all workers in a master-worker program are forced to deal with a single "task manager" process when they need work assignments. It can complicate the source text, forcing workers to seek out the appropriate manager rather than performing simple data manipulations directly.

4.3 Cheaper, faster and more expressive synchronization

Programming experience makes it clear that the tight synchronization imposed by current Ada's rendezvous is often (if not usually) unnecessary. Ada forces a sending process to block until its message has been accepted by a receiver. In fact, it is common for processes in a parallel application to generate data without any need for an acknowledgement or a reply. In Ada, these sending processes block needlessly, forcing the system to reschedule or--in the absence of other runnable jobs on the sender's node--simply to idle, for no logical reason. Using Linda operations, the data in question is added directly to tuple space, and the sender continues immediately. Returning to master-worker programs specially: workers must not be forced to use blocking communication protocols unnecessarily--in a well-structured master-worker application, they will be active almost continuously.

4.4 Flexible naming

Current Ada forces a sending process to send data to some particular recipient. Programming experience makes it clear that it's common for processes in a parallel or distributed application not to care (or not to be able to predict) which process will "receive" the data they generate. This holds particularly for master-worker programs, where the workers are identical and, mutually anonymous. Inter-worker communication and master-worker communication is necessary whenever tasks need to be distributed or current status information made available. But current Ada's rendezvous is an unacceptable protocol here: workers are identical, and tend to pick up tasks (and hence short-term "identities") dynamically. Thus, it tends to be impossible to single out which worker to send some data object to. Workers must themselves grab what they need--easily accomplished in Ada Lite, difficult or impossible in current Ada.

4.5 Flexible Granularity

Linda-extended Ada would support applications along the full granularity spectrum--relatively fine-grained computations for parallel processors through coarse-grained applications on LANs (and even WANs). (For example, we discuss in [CG89] experiments with an application that shows good performance on the Intel iPSC/2 hypercube, and on shared-memory multiprocesors; increasing the grain size by a factor of 0--a change to one line of the code--gives us a very coarse-grained application that runs well on eight Ethernet-connected Unix workstations. See also [WL88], for discussion of a Linda application that run faster on fourteen networked VAXes than on a Cray.)

4.6 Integrated support for databases

Linda blends naturally into database systems: Linda's tuple space can be regarded as a form of synchronizing, sharable relational database. Current research is directed at supporting persistent tuple spaces directly on commercial relational database systems. We believe that strong, integrated support for database applications will be crucially important in future, as sophisticated, first manipulation of very large databases becomes critical in a wide variety of domains.

4.7 A natural basis for heterogeneous and dynamic systems

Ada Lite is also, potentially, a better basis for heterogeneous or evolving computations (sometimes cailed "open systems") than is Ada. Because the Linda operations can be added to any host language, Linda's tuple space can serve as the basis for inter-process communication between modules in different languages. Furthermore tuple space, but not current Ada's rendezvous, supports communication between mutually unknown or time-disjoint processes--in fact allows such processes to cooperate in building the same data structures. Invoking a service by adding an object to tuple space (rather than by calling a remote entry)

makes it possible for any currently-appropriate process to pick up the tuple and perform the service. Tuple space automatically and implicitly buffers communication between computations taking place at different rates, in different places or during disjoint lifetimes. Supporting heterogeneous computations in Linda is a main focus of our current research.

4.8 Conclusions

In a note called "The Simplicity of Linda", Steven Zenith of Inmos summarizes the case as follows:

> Let's use some buzz words to describe the Linda paradigm. This is a) fun and b) helps us to understand why Linda is liked so much. So Linda features:
>
> easy to use concept
>
> automatically scalable algorithms
>
> portable applications programming (architecture independent)
>
> the same model on different parallel architectures
>
> a unified process and communications model
>
> an integral persistent storage concept
>
> Linda is easily added to existing sequential languages to create parallel variants of the original language (e.g., C-Linda, Lisp-Linda, Postscript Linda, Modula-Linda exist currently).

We agree. Elsewhere Zenith writes that "Linda is elegant and easy to understand"; Professor K. Birman of Cornell calls Linda "the most elegant piece of work in the area". Elegance means power with simplicity, and these attributes, to summarize, make Ada-Linda a promising vehicle for distributed and parallel programming.

REFERENCES:

"Building an Ada-Linda Parallel Programming Environment." SCA TR 148. Submitted as a Phase I proposal to SDIO (January 1989).

N. Carriero and D. Gelernter, "Applications Experience with Linda," in Proc. ACM Symp. Parallel Programming, (July 1988).

D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel programming in Linda," in Proc. Int. Conf. Parallel Processing, (August 1985).

R.A. Whiteside and J.S. Leichter, "Using Linda for supercomputing on a Local Area Network." in Proc. Supercomputing '88 (November 1988).

SUPPORT FOR INHERITANCE AND POLYMORPHISM

DATE: October 31, 1989

NAME: Lennart Mansson

ADDRESS: Box 4148
S-203 12
Malmo, Sweden

TELEPHONE: +46-40-25 46 36

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

As was pointed out at the Destin Requirements Work Shop for Ada 9X [Ada 9X Project Report, June 1989] there are problems with maintenance of large Ada programs over a long period of time.

In many situations you have need in your software to represent many "objects" with similar behavior but different in some respect. They may have to execute different algorithms in certain situations, or they have different internal data structure, or some of them may be tasks and must respond to some extra entry, etc.

Today the way of solving such problems is to use variant records, case statements and/or some kind of generic packages. Those approaches lead to problems. Often one or more enumeration types, enumerating all the similar variants, are essential for the design of the system, leading too huge recompilation when a new variant has to be added. In a continuously operated system, recompilations definitely makes it impossible to develop the system incremental. System wide change analysis must also be applied in order to guarantee that case statements still are correct.

If one tries to avoid the central enumeration type, often several types are defined that have lots of properties in common, but must be handled as separate types. The cost of having multiple copies of most of the code of several objects cannot be overestimated.

The need for keeping references to several variants in the same lists or call entries with the same parameters may have to be solved with unchecked_conversion. Generics saves a lot of code but generally does not solve the basic problem.

The central enumeration type and unnecessary copies of code is not acceptable in a large system. Future development of the system often includes adding new variants. This operation should only require changing the modules of the system that really has to care about the new variant.

Other related problems are handling of variants of software components which mostly is solved by copying and changing a little.

Object oriented programming languages (OOPL) has solved those problems with the class hierarchy, inheritance schema and an ability to treat objects of different types with the same code. (The later property is often referred to as polymorphism.) But an OOPL often lack real time facilities, strong typing modules other than objects, specifically not modules exporting types, exceptions or nested modules and all the other advantages that are well known benefits of Ada. Most OOPL's only provides a flat structure of objects.

We believe it is possible to extend Ada in a very natural way to allow for incremental development, via inheritance, in the way that has proven successful in object oriented languages as Simula [Simula 87] and its successors. The main idea is to make the central enumeration types unnecessary and instead have them implicit as different subclasses in a hierarchy of object types.

The important thing is to do this on the terms of Ada and not to impose concepts from other languages without seeing how they fit into Ada. Instead concepts already in Ada can be generalized and combined with the old. The result is not necessarily a typical object oriented language but something that benefits from both. Especially the combination of object orientation and parallel processing and modularity is very important and challenging.

The required solution should support inheritance and polymorphism for "objects". Nevertheless should strong typing been forced, allowing compile time checks of the applicability of operations.

The solution does should not lead to bad run time performance. There are solutions in existing Object Oriented Languages (c.f. Simula) that supports polymorphism at the cost of one indirect addressing instead of a direct.

It is also important that inheritance and polymorphism can be applies on tasks.

IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

Those mentioned in the problem statement, and in some cases use of unchecked conversion between types that have some common structure.

POSSIBLE SOLUTIONS:

The solution should consider three Ada concepts as "objects":

Packages
Records
Tasks

Packages need not be first class objects (i.e., dynamic allocation, passing as parameters etc) but it must be possible to declare a new package as a specialization of an old, adding (visible or private) declarations.

It is also important to be able to override the body of subprograms in the body of the base package when specializing a package. (This can be restricted to subprograms that are marked as "redefinable" in the base package.)

Inheritance and specialization on package alone gives the fundamental needs for maintaining abstract, non typed, objects through revisions and variants. (of requirement 3.III.B.1..1 in the Ada 9X Requirements Workshop Report).

Record definitions can be the base for passive typed objects by allowing a new kind of subtyping by adding new components. [c.f. Wirth, N:."Type Extensions", ACM Transactions on Programming Languages and Systems, Vol 10, No2, April 1988, pp 204-214]

The combination of specialization of packages and specialization of records and extending overloading resolution to subtypes of the new kind gives an abstract data type concept that covers all the required functionality.

An alternative approach would be to introduce package types and specializations on them. Those types would have to first class types in order to allow for full required functionality. But, then specialization of records and extended overloading would not be needed.

For tasks specifications it is natural to allow for a subtype concept where new entries can be added when declaring a subtype. For the corresponding bodies two alternatives should be available. The first is total redefinition of the body. In the second code defined in the subtype definition can be put in one or more "place holders" declared in the base type (e.f. for the Inner-concept of Simula-67). In this second alternative local declarations in the body of the base should be inherited to the subclass.

If specialization of tasks and/or records is introduced as subtypes, it is also natural to introduce subtypes of access types with a "subtype constraining", i.e, constraining the access object to denote a certain subtype and all subtypes based on that one.

WHEN/RAISE CONSTRUCT

DATE: October 19, 1989

NAME: James Lee Showalter, Technical Consultant

DISCLAIMER:

This Revision Request has been submitted to some amount of peer review by others at Rationale, but it is not an official Rational submission.

ADDRESS: Rational
3320 Scott Blvd.
Santa Clara, CA 95045-3197

TELEPHONE: (408) 496-3706 (11 a.m. -- 9 p.m.)

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

Assertion checkers are often implemented as a set of IF statements which, if the condition evaluates True, raise some exception:

```
if Some_Condition then
        raise Some_Exception;
elsif Some_Other_Condition then
        raise Some_Other Exception;
end if;
```

These conditionals contain a lot of syntactic sugar that makes reading and/or debugging them somewhat difficult (not to mention the difficulty of typing them in correctly in the first place).

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Add a WHEN/RAISE construct to the language. Using this construct, the above code sample would be rewritten as follows:

```
when Some_Condition raise Some_Exception;
when Some_Other_Condition raise Some_Other_Exception;
```

COMPATIBILITY:

The proposed solution is upward-compatible. All previously-compiled code will re-compile successfully and will behave identically during execution except for possible small changes in execution speed.

FEATURES NECESSARY TO SUPPORT APPLICATIONS IN CONTROL ENGINEERING

DATE: October 18, 1989

NAME: Wolfgang A. Halang

ADDRESS: University of Groningen
Department of Computing Science
P.O. Box 800
9700 AV Groningen
The Netherlands

TELEPHONE: +3150633939 (secretary)
+3150633925 (extension)
+3150633976 (telefax)
E-mail: hp4n1!guvaxin!halang.UUCP

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

The following features are necessary to support applications in control engineering (see "Adaptation of Ada to the Requirements of Industrial Control Problems" Proceedings of the 10th IFAC World Congress on Automatic Control, Vol. 4, pp. 96-101, Munich, 27-31 July 1987. [attached] for more detail):

1. Additional Language Elements in Ada

2. Software Verification Features

ADDITIONAL LANGUAGE ELEMENTS FOR ADA

When comparing the requirements for real-time languages and systems, with the capabilities of Ada, it becomes obvious that various elements especially important for the production of reliable software are still missing, and that some changes of Ada's semantics are necessary.

First, statements allowing the complete controllability of tasks from the outside in dependence on complex schedules involving interrupts, time specifications, and further conditions should be introduced into Ada.

For encapsulating the access to protected resources, we therefore introduce a lock statement, similar to the one introduced by Barnes (1979).

TABLE Additional Real-Time Features

- Application oriented synchronization constructions
- Time-out of resource claims
- Availability of current task and resource stati
- Inherent prevention of deadlocks
- Feasible scheduling algorithms
- Early detection and handling of overload

- Determination of entire and residual task run-times
- Exact timing of operations
- Application oriented simulation regarding the operating system overhead
- Event recording
- Tracing
- Usage of only static features if necessary

SOFTWARE VERIFICATION FEATURES

A step toward enabling a technical safety approval of real-time programs is to create the possibility for application oriented simulation.

Given the possibilities for event recording and tracing, no additional features need to be introduced into Ada to facilitate the simulation of application software, because the language already contains the statements necessary to generate external and internal events, viz. the entry call and the raise statement.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

See "Adaptation of Ada to the Requirements of Industrial Control Problems" Proceedings of the 10th IFAC World Congress on Atuomatic Control, Vol. 4, pp. 96-101, Munich, 27-31 July 1987.

POSSIBLE SOLUTIONS:

See "Adaptation of Ada to the Requirements of Industrial Control Problems" Proceedings of the 10th IFAC World Congress on Automatic Control, Vol. 4, pp. 96-101, Munich, 27-31 July 1987.

PACKING VARIABLE LENGTH RECORDS INTO ONE BUFFER FOR TRANSMISSION

DATE: June 6, 1989

NAME: Robert R. Van Tuyl

ADDRESS: GTE
P.O. Box 7188 M/S 7G32
Mountain View, CA 94039

TELEPHONE: (415) 966-4024

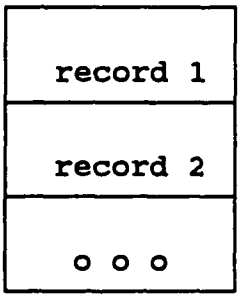
ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

There does not seem to be an efficient way to incrementally construct a message of variable size records. That is, as the data is gathered, it should be placed in the exact place in the message buffer for transmission. The data is gathered in the order that it is placed in the records. Each record is of arbitrary length determined by the content of the data so the buffer cannot be laid out in advance either at compile time or run time.

Each record is an individual message. They are grouped together to increase the bandwidth during transmission (fewer overhead bytes and acknowledgements). The timing is critical.

The record usually contains at least one variable length component (an array). The record has other components besides the variable length component.



IMPORTANCE: ESSENTIAL

CURRENT WORKAROUNDS:

The current workaround is to use a component in the record to indicate the size of the variable component. Then unchecked conversion is used to convert addresses to access types and vice versa. In a linear address space, addresses convert nicely to access types (with some care to avoid taking addresses of descriptors). Address calculations are used to find the beginning of each record and then the address is translated into

an access type pointing to the record. The address manipulations are hidden from the casual user in a package, all the user sees is the access type (This mitigates the erroneous Ada but does not excuse it).

POSSIBLE SOLUTIONS:

Clearly, one could add features to the language to allow for explicit packing and unpacking of variable size records. I would argue for a somewhat kludgier solution.

Add an attribute of an object next_locations_address_after_object. This attribute returns the address of the memory location the follows the last memory location used by the object. This is somewhat tricky to compute for a complex record, although it can be done.In some cases, one can obtain this address simply by placing a null component at the end of the record and finding the address of the null component.

Allow the discriminant which specifies the size of the last array component to be changed during or after the adding of data to the record.

Allow for explicit language supported conversion of an address to an access type and vice versa.

With these features the construction would be more or less as follows:

- Obtain pointer to first record in buffer (convert address to access type)
- While end of current record<= end of buffer loop -- using next location attribute
    - Fill record with data, counting number of items placed in record variable length component.
    - Change the discriminant for the variable length component (also the last component)
    - Get a new access type object using the next location address attribute and the explicit conversion to an access type.
    - Exit if there are no more messages
- end loop
- Send the data that has been placed in the buffer. The current value of the access type object is just beyond the last location to send.

Unpacking the data is achieved in a similar fashion. Each message is processed as an individual record.

GENERAL LANGUAGE RECOMMENDATIONS

DATE: July 14, 1989

NAME: J. Yost

ADDRESS: GTE
PO Box 7188
M/S 7G32
Mountain View, CA 94039

TELEPHONE: (415) 966-4024

ANSI/MIL-STD-1815A REFERENCE: NONE

PROBLEM

Editor's Note: The following are general Ada language recommendations.

Memory Allocation

Recommend changes which would permit the implementation of the "NEW" function or its equivalent. Ada currently cannot implement a generalized memory management routine, due to typing constraints.

Tasking

Recommend moving all tasking functionality into a library package (as I/O is implemented int he TEXT_IO, DIRECT_IO, etc. library packages) Ada currently incorporates tasking as a language entity. Some language hooks would have to be provided to identify tasks, but all synchronization, communication, creating, destruction, etc., would be handled through the library package. In this way the distinction between the host operating system tasking capabilities and the Ada capabilities would be lessened.

Library Unit Visibility

Recommend library extensions that permit the restriction of library unit visibility. This permits the creation of "hidden" library units which may be used by selected library units, but not others. This is analogous to the Rational "subsystem" concept.

Separate Compilation of Overloaded Subprograms

Recommend library extensions which permit overloaded subprograms (i.e., subprograms with the same name) to be separately compiled. Currently, libraries will not allow two entities of the same name, which implies that one must include all overloaded subprogram bodies with the body of a containing package, rather than separately compiled.

Exception Context

Recommend that access to the"context" of an exception be available at run-time to an exception handler. by the time an exception handler gets control, the context information about the unit switch raised the exception has disappeared (popped from stack, etc.). There is valuable information for debugging and recovery (the calling tree, the value of data at the time of the exception, etc.) which should be available to the exception handler.

Exception Alias Handling

Recommend that Ada allow the handling of exception aliases within the same exception handler. Ada allows exceptions to be renamed. Thus, an exception handler might wish to handle an exception from two different packages (with different names) which are in fact the same base exception. Ada will not allow both names to be specified, as it knows that they are aliases.

Anonymous Exceptions

Recommend that Ada allow the exception handler to acquire the name of an exception captured through an OTHERS handler. Ada does not allow an OTHERS handler to acquire the name of the exception being handled.. This prevents generalized exception handling, or at least printing useful debugging messages.

More Detailed Definition of Standard Exceptions

Recommend that standard exceptions be defined at a finer granularity. Ada defines an exception called STORAGE_ERROR, which may mean that the heap is out of memory, an illegal memory reference has occurred, etc. These are very different kinds of errors.

Creation of Secondary Standards

Recommend that secondary standards be created to document the relationship between the language and outside constructs. These standards would be independent of the language standard. Examples are industry standards bindings (X-Windows, SQL, GKS, etc.) and library packages (currently TEXT_IO, etc.) but to include extensions such as math, statistics, resource locking, etc., packages.

Generalize Generics

Recommend that restrictions on generics be removed to make them usable with all Ada constructs. For instance, a generic cannot be instantiated with an exception as a generic parameter.

Allow subprogram Parameters

Recommend that subprograms be considered types to allow passing as parameters to there subprograms, assignment to variable, etc. This would allow more flexible interface to other programming languages, such as interfacing to the X-Windows Library in C.

Pragma Interface in Body

Recommend permitting the use of pragma interface within a body, as well as a specification. Ada permits the mapping of the definition of a routine to a "foreign" routine through PRAGMA INTERFACE. Since this must appear in the specification, the mapping is not hidden as a design detail.

Renames in Body

Recommend permitting the use of procedure RENAMES within a body, as well as a specification. Ada permits the renaming of a routine RENAMES. Since this must appear in the specification,k the mapping is not hidden as a design detail.

Task Cleanup on Termination of Parent

Recommend that when a terminate alternative in a select statement is taken, task cleanup code, which can be located between the terminate statement and the end select statement is permitted to be executed. This will eliminate the need for a separate task entry pint to cleanup after the task once the parent unit has reached its end. This will also prevent thoughtless programmers from "forgetting" to clean up any resources after a task is no longer needed.

IMPORTANCE:

CURRENT WORKAROUNDS:

POSSIBLE SOLUTIONS:

CONSTANT (AND STATIC) FUNCTIONS

DATE: June 9, 1989

NAME: Bryce M. Bardin

ADDRESS: Hughes Aircraft Company
Ground Systems Group
P.O. Box 3310, M/S 618/M215
Fullerton, CA 92634

TELEPHONE: (714) 732-4575
E-mail: BBardin@ajpo.sei.cmu.edu

ANSI/MIL-STD-1815A REFERENCE:

PROBLEM:

It is often desirable to return a constant and statically defined value as an attribute of a class of similar data types. Because the data types are similar, all of these values should have the same (overloaded) name, so the value should be a function. In order for it to be usable as a primary in a static expression, it must be considered static. For efficient implementation it should be recognized as constant.

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS: NONE

POSSIBLE SOLUTIONS:

Allow constant functions which always return the same static value to be declared which would be allowed as primaries in static expressions.

For example:

```
package Vectors is
        type Index is (X, Y, Z);
        type Vector is array (Index range <>) of Float;
        subtype V1 is Vector(X .. X);
        subtype V2 is Vector(X .. Y);
        function Unit_Vector return constant V1;
        function Unit_Vector return constant V2;
end Vectors;

package body Vectors is
        function Unit_Vector return constant V1 is
        begin
                return (1.0); -- expression must be static
        end Unit_Vector;
        function Unit_Vector return constant V2 is
```

```
        begin
                return (0.707, 0.707); -- expression must be static
        end Unit_Vector;
end Vectors;
```

REVISION REQUESTS THAT WERE SUBMITTED AS STUDY ADA COMMENTARIES

SECTION 17.

Allow DATA of mode "in" in SEND_CONTROL DRAFT 84-03-26 AI-00003/01 1

| !standard 14.06 (05) 84-03-26 AI-00003/01
| !class study 84-03-26 (provisional classification)
!status work-item 83-10-07
!topic Allow DATA of mode "in" in SEND_CONTROL

| !abstract 83-10-07 (DRAFT)

SEND_CONTROL should have an IN DATA parameter (rather than IN OUT).

| !discussion 83-10-07 (DRAFT)

Often the second argument of SEND_CONTROL will be a constant. A constant cannot be written directly at present because the parameter mode is IN OUT. Either the parameter mode should be mode IN, or, if SEND_CONTROL is expected to receive as well as transmit information, separate parameters should be used for these purposes.

!appendix 83-10-07

| **********

!section 14.06 (05) William R. Greene 83-03-29 83-00002
!version 1983
!topic : Allow DATA of mode "in" in SEND_CONTROL

Often the control information appropriate for passing to a device as the second argument of SEND_CONTROL is a constant. Mode "in out" is inappropriate for such an argument. I suggest the addition of the following line :

```
procedure SEND_CONTROL (DEVICE: in device_type; DATA: in data_type ) ;
```

to the outline of package LOW_LEVEL_IO.

Allow -1..10 as a discrete range in loops 85-09-16 AI-00140/01 1

| !standard 03.06.01 (02) 85-09-16 AI-00140/01
!class study 84-02-06 (provisional classification)
!status received 84-02-06
!topic Allow -1..10 as a discrete range in loops

!abstract 84-03-16

The rule disallowing ranges such as -1..10 in iteration rules, constrained array type declarations, and entry family declarations should be changed to allow any expression having the type universal_integer as both bounds of the range.

!recommendation 84-03-16

!wording 84-03-16

!discussion 84-03-16

| !appendix 85-09-16

****

!section 03.06.01 (02) J. Goodenough 83-11-18 83-00211
!version 1983
!topic legality of -1..10

The rule in this paragraph makes

for I in -1..10 loop

illegal when an implementation has more than one integer type declared in STANDARD and legal when an implementation has decided to support only one predefined integer type.

To see this, consider the case where INTEGER is the only type declared in STANDARD. 1 (and 10) are either of type universal integer or can be implicitly converted to any integer type in scope, namely INTEGER and COUNT (which is declared in TEXT_IO) (the implicit conversion to COUNT is a red herring, so don't worry too much about it). We now ask what unary "-" operators are visible that take operands of type universal integer, INTEGER, and COUNT, and find that only the operators for universal integer and INTEGER are visible. (since we have not said "with TEXT_IO; use TEXT_IO"). Hence, the bounds of the range can be either of type universal integer or INTEGER. We now apply the first sentence of 3.6.1(2):

... an implicit conversion to the predefined type INTEGER is assumed if

> each bound is either a numeric literal, a named number, or an attribute, and the type of both bounds (prior to the implicit conversion) is the type universal integer.

Although -1 can have the type universal integer, it is not a numeric literal. Since the expression -1 fails the first criterion, this part of the rule does not apply. We now go on to the rest of the rule:

> Otherwise, both bounds must be of the same discrete type, other than universal integer; this type must be determinable independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type.

The first phrase says the bounds cannot be of type universal integer. This leaves only the possibility that the bounds are of type INTEGER. Since both bounds have the same discrete type, and the type has been determined independent of the context, -1..10 is legal, and the bounds are of type INTEGER.

Now if LONG_INTEGER were also declared in STANDARD, then the second part of the rule allows the bounds to be either of type INTEGER or LONG_INTEGER. Since the type of the bounds cannot be resolved, -1..10 is illegal.

In short, the legality of -1..10 depends on whether an implementation has chosen to implement more than one integer type! This was surely not the intent and is, to my way of thinking, even more repugnant than having -1..10 illegal when there are two integer types declared in STANDARD.

I believe the rule was intended to read:

> if the type of both bounds is universal integer prior to any implicit conversion, then each bound must be either a numeric literal, a named number or an attribute, and an implicit conversion to the predefined type INTEGER is assumed. Otherwise, ... [as before]

Although this was probably the intent (see comment #3705), there really would have been no harm in allowing -1..10 to be legal by providing a preference for an implicit conversion to INTEGER in this case. The rule probably should make F..G illegal if both F and G are user-defined functions overloaded to produce more than one discrete type; the preference rule should apply only to expressions that can be of type universal integer. Since such expressions can also be of type INTEGER (either by direct implicit conversion or by implicitly converting some operand lower in the syntax tree), it is sufficient to say that the type of the expression is assumed to be INTEGER when the preference rule is to be applied. A possible rule would be:

> For a discrete range used in a constrained array definition and defined by a range, if each bound is allowed to have the type universal integer and the type predefined INTEGER (when other rules of the language are considered), then both bounds are assumed to be of type predefined INTEGER. Otherwise, ...

In short, making -1..10 legal when only one integer type is defined by an implementation is a mistake that should definitely be corrected. It can be corrected either by making such a range always illegal (in the contexts specified currently by the RM), or preferably, by making the range always legal.

****

!section 03.06.01 (02) 8.6(2) P. N. Hilfinger 83-11-19 83-00216
!version 1983
!topic legality of -1..10
!reference 83-00211

I second John's suggestion that we change 3.6.1(2) to fix the unintentional implementation dependency and to make -1..10 legal. I believe that the illegality of the latter is also unintentional, since the rationale given for the change that made this illegal concerned only ranges used in arrays. (It said something to the effect that arrays dimensioned, e.g., -1..10 aren't all that useful anyway. For the record, I believe that contention is probably false too.)

Others have pointed out that technically, the problem is even worse than this. According to 8.6(2), some incompletely determined subset of the library units in the library are in scope for any compilation unit (they aren't visible, however.) This means that in fact, even if there is only one integer type in STANDARD, the range -1..10 MAY be ambiguous (under the current wording) if there is a library unit containing a numeric type declaration (possibly one that is not WITHed either directly or indirectly by any compilation unit involved in the entire program!)

****

!section 03.06.01 (02) J. L. Gailly 83-11-21 83-00218
!version 1983
!topic legality of -1 .. 10
!reference 83-00211

The intent was that -1..10 is always illegal in this context. The only rules you can apply to resolve the overloading are given explicitly: the type must be dicrete and both bounds must have the same type. In particular, you cannot use the fact that the type must not be universal_integer, so you have at least two candidates (universal_integer and INTEGER) and consequently -1..10 is ambiguous.

I agree that this result is not desirable (it is still better than having to count the number of integer types in scope). However, all proposed solutions seen so far are ambiguous or contradictory with 4.6(15). The

proposal made by Taffs in #3705 and #5435 requires oveloading resolution in two stages: first, try to resolve without any special rule; then if this failed, apply an implicit qualification (NOT a conversion) by the type INTEGER (this is my interpreation of his sentence "the wording must provide a particular context, rather than apply a particular implicit conversion"). Taffs is right in saying that his proposed rule does not add complexity to the implementation of overloading resolution (one bottom-up then one top-down pass is still sufficient), but the two passes are logically different. Taffs' solution also resolves examples that were intended to be ambiguous (see #3705), but these examples are pathological.

John G. also suggests a preference rule for INTEGER when only universal_integer and INTEGER are candidate types: this rule does not seem to solve the problem of the presence or absence of LONG_INTEGER. (May be he meant "if each bound is allowed to have a predefined integer type ...", but that would also resolve arbitrarily ambiguities created by the user with explicit redefinitions of operators.) It is also not clear whether "assumed to be" means implicit qualification (as in Taffs' solution), implicit conversion (in contradiction with 4.6(15)), or is a new concept in Ada.

I believe that a solution using implicit qualification could work (although it leads sometimes to unexpected results), but I don't have a good wording to suggest. It is also possible to modify 4.6(15), saying that 3.6.1(2) is the only case where the rule defining convertible operands does not apply. 3.6.1(2) would then say "if both bounds are of type universal_integer, then each bound is implicitly converted to the predefined type INTEGER".

The two solutions (implicit qualification or implicit conversion) yield different results only for pathological examples; the solution using conversion is closer to what most users would intuitively expect.

****

!section 03.06.01 (02) J. Goodenough 83-11-21 83-00219
!version 1983
!topic legality of -1..10
!reference 83-00216

The problem is really independent of 8.6(2), since even if more than one integer type is technically in scope because it is in some library unit that is not with'd, the unary minus operator for that type will not be visible, and so the additional integer type creates no ambiguity. In essence, the final determiners of ambiguity are the visible operators, not the types that are in scope.

****

!section 03.06.01 (02) J. Goodenough 83-11-22 83-00220
!version 1983
!topic legality of -1..10

!reference 83-00218

My proposed wording did not say that "when only universal_integer and INTEGER are candidate types"; it said, in effect, when universal_integer and INTEGER (at least) are candidate types. Hence, if LONG_INTEGER is also a candidate, the resolution to INTEGER is chosen.

My proposed wording is equivalent to an implicit qualification of the bounds when there is otherwise no way to choose. My phasing "assumed to be" was suggested by the current phrasing; how about saying "then both bounds are required to be of predefined type INTEGER." I don't really care whether the bounds are disambiguated by qualification or by implicit conversion from universal integer, since as #3705 shows, only highly pathological cases are likely to be affected.

My proposed rule does not contradict 4.6(15) since it specifies a condition under which the result type universal_integer is not allowed, but for optimal clarity, 4.6(15) should also metntion the rule in 3.6.2.

****

!section 03.06.01 (02) Peter Belmont / Intermetrics 83-11-29 83-00228
!version 1983
!topic legality and meaning of -1..10

This question is one which begs, I think, for a statement of REQUIREMENTS on the design and/or interpretation. For myself, I would state the requirement as one of simplicity, completeness and non-surprisingness, and parallelism with the other Ada construct which presents a similar problem, (-1<10).
I would like "-1" to be treated as a literal in contexts which do not force conversions to non-universal types.

Now, how do we treat

(-1 < 10) ?

As I understand it, we see that it is possible to understand this without applying implicit conversions. That is, we treat the "-1" as a universal integer, as a literal.

I propose that we seek to restate the LRM to achieve the same effect. Treat

exp1 .. exp2

as a construct that allows an interpretation in universal integers. And then, in a post-processing step, convert to INTEGER.

If you like this, one could re-write the paragraph:

> A discrete range defined by a range and used
> in a constrained array definition (see 3.6),
> in an iteration rule (see 5.5), or in the declaration
> of a family of entries (see 9.5) is subject to the following
> rule.
>
> The two bounds must be of the same discrete type, which
> may be universal integer. If
> they are both of type universal integer, then an implicit
> conversion of both bounds to the predefined type INTEGER
> is applied after the bounds are elaborated.

Discussion. First, this makes a good parallel with (-1 < 10). Second, it handles -1..10 as we intuitively expect it to be handled. Third, in a case like

    -1 .. INTVAR

an implicit conversion of the 1 to INTEGER (allowing an INTEGER interpretation of the -1) is always available (see 4.6(15)) in any case. Fourth, we leave no holes (that I can see). Fifth, non-static universal values (from T'LENGTH(2)) are OK; conversion to INTEGER may raise an exception at run-time.

Question: Why does the LRM place such an odd restriction on the expressions which will be converted to INTEGER? What important interest is served by providing a friendly home for 1..10 but a cold winter wind for -100..-1? Why should I get in trouble for writing 0..T'LENGTH-1 when I am allowed 1..T'LENGTH without demur?

****

!section 03.06.01 (02) Jean D. Ichbiah 84-03-01 83-00309
!version 1983
!topic legality of -1 .. 10
!reference 83-211 and 83-218

I agree with comment 83-218 (and disagree with 83-211): The following declaration is illegal

    A : array(-1 .. 10) of CHIPMUNK;

The reason - as pointed out by JL Gailly - is given by the sentences:

> (a) Otherwise, both bounds must be of the same discrete type, other than universal_integer;

(b) this type must be determinable independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type.

So, for overload resolution of the range -1 .. 10 we proceed as follows:

. First we cannot apply the first sentence of 3.6.1(2) since "-1" is not a literal (so far 83-211 is correct).

. Hence we try to apply the second sentence (reproduced above). But this yields at least 2 possible interpretations of the subsentence (b):

- The operator "-" in "-1" is the unary operator that applies to universal_integers. Both bounds are of type universal integer.

- Apply implicit conversions to INTEGER to the two literals "1" and "10" then the INTEGER "-" operator.

. Consequently there are two possible types and the type is not uniquely determinable for the context. (Note that (b) excludes the use of (a) for this determination.)

Hence the statement given in 83-211 (that legality is implementation_dependent) is incorrect.

Concerning functionality, I do not see any reason for a change:

(a) Positive ranges are quite common and are well covered by the rule.

(b) Negative ranges are very rare: they already assume a sophisticated programmer, and in such a case the available tools are sufficient:

```
A : array(INTEGER range -10 .. 10) of ...
for N in A'RANGE loop ...
for N in INTEGER range -10 .. 10 loop ...
```

(c) Why make the rules more complex?

****

!section 03.06.01 (02) RD Pogge/Naval Weapons Ctr 85-08-14 83-00641
!version 1983
!topic Invalid bounds

The first sentence of 3.6.1, paragraph 2, contains a long list of bounds converted from Universal_Integer to Integer. However, it has been interpreted by at least one validated compiler that -1..5 is an invalid construct because -1 is a

simple expression of type Universal_Integer and 5 is a numeric literal which is converted to type Integer, so the bounds are of different types. We recommend a rewording of 3.6.1(2) to be:

...a numeric literal, a simple expression, a named number...

The following:

"for i in -10..10 loop"
"is array (-1..1) of.."

should be legal, and the validation test suite should check to make sure that they are.

Proposed solution to packed composite object a DRAFT 84-08-27 AI-00142/02 1

| !standard 09.11 (04) 84-08-27 AI-00142/02
!standard 09.11 (05)
!standard 09.11 (10)
!class study 84-07-13 (provisional classification)
!status work-item 84-01-18
!references AI-00004
!topic Proposed solution to packed composite object and shared variable problem

!abstract 84-01-18

The problems discussed in AI-00004 may be resolvable by restricting the non-erroneous shared access to arrays and extending the SHARED pragma.

!recommendation 84-01-18 (DRAFT)

The restrictions in section 9.11(4,5) should be extended to apply to all types. Correspondingly, the SHARED pragma should be extended to all types. The interpretation of SHARED applied to a composite object should be that shared access to distinct components of the object are proper and that the components of the object are themselves SHARED (but shared access to the object, as in simultaneous assignment, is erroneous as currently.) SHARED should also be applicable to types, with the same restrictions as for the PACKED pragma, meaning that all objects of the type are SHARED (and in the case of access types, that all accessed objects are shared).

!discussion 84-01-18 (DRAFT)

As indicated in AI-00004, there is an implementation problem with 9.11(4), in that it always allows the individual fields of a composite object to be accessed independently by two tasks, regardless of the representation of the object. This presents problems with packed composite objects.

One possible solution to the implementation problem is to return to a former wording and extend the provisions of 9.11(4) to all objects. Thus, it would be erroneous for two tasks to access the same object (regardless of type)--or any subcomponents of that object--between synchronization points.

This, of course, introduces programming problems. For example, programs in which n tasks each handle a row of a matrix become erroneous. To alleviate the problems, we can extend the SHARED pragma. For a composite object, T,

```
pragma SHARED(T);
```

means that each access to a scalar or access typed subcomponent of T is a synchronization point.

Finally, to allow for unnamed objects (those accessed through access types), the SHARED pragma could be extended to access type names, meaning that each object pointed to by a value of that access type would be shared.

!appendix 84-08-27

****

!section 09.11 (10) Norman Cohen 84-6-25 83-00403
!version 1983
!topic Local copies of shared subcomponents

This paragraph forbids the application of the Shared pragma to a scalar- or access-type subcomponent of a shared variable (even if that subcomponent has been renamed as a variable).

However, paragraphs 3.2(7) and 3.2.1.(3) establish that a subcomponent of a variable is a variable. Therefore, 9.11 permits an implementation to keep a local copy of a scalar- or access-type subcomponent of a shared variable. There is no way for the programmer to mandate use of the shared subcomponent itself rather than the local copy.

Additional control statement for use w/in 84-03-13 AI-00211/00 1

!standard 05.07 (00) 84-03-13 AI-00211/00
!class study 84-03-13 (provisional classification)
!status received 84-03-13
!topic Additional control statement for use w/in a LOOP statement.

!abstract 84-03-13

!recommendation 84-03-13

!wording 84-03-13

!discussion 84-03-13

!appendix 84-03-13

****

!section 05.07 F. Six, SINGER 83-02-28 83-00319
!version 1983
!topic Additional control statement for use w/in a LOOP statement.

A statement

```
continue_statement ::=
    continue [loop_name] [when condition] ;
```

would be very useful in Ada in avoiding excessive nesting levels which cause programs to be unreadable. The continue should have the effect of causing a go to to a labeled null statement immediately preceding the end loop [loop_name].

Refer to Kernighan, Ritchie, The C Programming Language, pp. 62 and 203.

Allow accept statements in program units 84-03-13 AI-00214/00 1

!standard 09.05 (08) 84-03-13 AI-00214/00
!class study 84-03-13 (provisional classification)
!status received 84-03-13
!topic Allow accept statements in program units nested in tasks

!abstract 84-03-13

!recommendation 84-03-13

!wording 84-03-13

!discussion 84-03-13

!appendix 84-03-13

****

!section 09.05 (08) M. Tedd 84-02-06 83-00275
!version 1983
!topic Allow accept statements in program units nested in tasks

It is desirable to allow accept statements inside program units contained within tasks. See my example below of a task which traverses a tree, accepting a rendezvous at each node.

Of course, it is desirable that the compiler be able to identify which task can obey an accept. I suggest allowing accepts within program units which are within the body of the task, but not in the body of nested tasks.

example;

```
task body TRAVERSAL is
    procedure TREE_WALK(TREE : ACCESS_NODE) is
    begin
        if TREE /= null then
          accept FIND_NEXT_ITEM (NO MORE : out BOOLEAN;
                                 PLACE : out ACCESS_ITEM)
                do
                   NO _MORE := FALSE;
                   PLACE := TREE.DATA;
            end FIND_NEXT_ITEM;
            TREE_WALK(TREE.RIGHT);
        end if;
    end TREE_WALK;
begin
    loop
```

```
                accept START;  --holds here until START is called
                TREE_WALK(ROOT);
                accept FIND_NEXT_ITEM (NO_MORE : out BOOLEAN;
                                       PLACE : out ACCESS_ITEM)
                    do
                        NO_MORE := TRUE;

                end FIND_NEXT_ITEM;
        end loop;
end TRAVERSAL;
```

Portability among machines w/ different c 84-03-13 AI-00216/00 1

!standard C (00) 84-03-13 AI-00216/00
!class study 84-03-13 (provisional classification)
!status received 84-03-13
!topic Portability among machines w/ different character representations.

!abstract 84-03-13

!recommendation 84-03-13

!wording 84-03-13

!discussion 84-03-13

!appendix 84-03-13

****

!section C F.Six, SINGER 83-04-28 83-00285
!version 1983
!topic Portability among machines w/ different character representations.

With some addition to the Ada STANDARD package it should be possible to port programs among machines with different character pepresentations and I think the goal is worth the addition. Application programs frequently require testing whether a given character is numeric, upper case letter, lower case letter, special, or control, and Ada could and should provide these in a fashion so that these programs are portable.

To this end, the package STANDARD could include several added enumeration types:

```
sub type NUMERIC is NUMERIC ('O',...,'9');
sub type LOWER is NUMERIC   ('a',...,'z');
sub type UPPER is NUMERIC   ('A',...,'Z');
sub type CONTROL is NUMERIC (nul,..., us, del);
```

which would permit one to use constructs such as char in NUMERIC to test and NUMERIC'POS (char) - 1 to convert from alphanumeric to integer.

Note, however, that the "subtype declarations" used here are not permitted within Ada, since enumeration is required rather than simply a range constraint. Thus these must be provided as part of the pre-defined language environment.

Resolution for the function CLOCK 84-03-13 AI-00223/00 1

!standard 09.06 (04) 84-03-13 AI-00223/00
!class study 84-03-13 (provisional classification)
!status received 84-03-13
!topic Resolution for the function CLOCK

!abstract 84-03-13

!recommendation 84-03-13

!wording 84-03-13

!discussion 84-03-13

!appendix 84-03-13

****

!section 09.06. (04) B.A. Wichmann 84-03-01 83-00321
!version ANSI
!topic Resolution fo the function CLOCK.

9.6.(4) states that DURATION'SMALL <= 0.020 (ie, 20 milliseconds). Although the function CLOCK produces a result of type TIME which gives access to the time in SECONDS, there is no guarantee that it will provide a resolution to 20 milliseconds. In fact, the validated DG/ROLM Ada compiler provides a resolution to one second. This resolution is not adequate for making some performance measurements. It is suggested that 9.6(5) is reworded to guarantee the 20 millisecond resolution of the function CLOCK. Alternatively, the resolution of CLOCK could SYSTEM'TICK, but there is no guarantee of the magnitude of this value at all.

As noted in a comment on an earlier draft of the standard, the problem above stems from an explicit capacity statement on the type DURATION. This is inconsistent with other basic types such as INTEGER and FLOAT which have no requirements on their range or precision. If the explicit properties of DURATION are to be specified, then this should relate to all the uses of DURATION in the language. In this case, it implies a statement on the resolution of the function CLOCK and the delay statement. Such statements can then be checked by tests in the validation suite. (Although the indeterminacy in tasking may make it difficult to test the actual delay in the delay statement.)

Real literals with fixed point multiplication 87-02-10 AI-00262/01 1

!standard 04.05.05 (10) 87-02-10 AI-00262/01
!class study 84-07-13 (provisional classification)
!status received 84-07-13
!topic Real literals with fixed point multiplication and division

!summary 84-07-13

!question 84-07-13

Should real literals be allowed as operands in fixed point multiplication and division?

!recommendation 84-07-13

!wording 84-07-13

!discussion 84-07-13

!appendix 87-02-10

****

!section 04.05.05 (10) P.Kruchten 83-06-31 83-00128
!version 1983
!topic can a real literal be an operand of a fixed point multiply ?

Query:
Can a universal_real be an operand of a fixed point multiplying operator ? Is 'universal_real' a case of 'any fixed point type' ?

or else:

Is there an implicit conversion of the real literal to one of the fixed point types visible at that point ? (LRM 4.6(15))

and then:

If there are several fixed point types visible at that point, shall the choice of the type be made on criteria such as the range or accuracy ?

Example:

```
procedure MAIN is
   type FX1 is delta 0.1 range -100.0 .. 100.0;
   type FX2 is delta 0.001 range -1.0 .. 1.0;
   A : FX1 := 5.0 ;
```

```
begin
   A := FX2( A * 0.001 ) ; -- type violation ?
                           -- ambiguous ?
                           -- equiv. to: FX2( A * FX2(0.001) ) ?
end MAIN;
```

****

!section 04.05.05 (10) P. N. Hilfinger 83-05-31 83-00129
!version 1983
!topic can a real literal be an operand of a fixed point multiply ? (83-00128)

It was the intent of the LDT (not clearly expressed in 4.10 and 4.5.5) that universal_real be a fixed point type for the purposes of 4.5.5.

****

!section 04.05.05 (10) Ron Brender 83-10-31 83-00201
!version 1983
!topic Can a real literal be an operand of a fixed point multiply?
!reference AI-00020

I concur with comment 83-00129 that it was intended that a real literal be allowed as an operand of a fixed point multiply (and division) operator. However, there is a subtle point that bears further consideration.

RM 4.5.5(11) states that

> "Multiplication of operands of the same or of different fixed point types is exact and delivers a result of the anonymous predefined fixed point type universal_fixed whose delta is arbitrarily small. The result of any such multiplication must always be explicitly converted to some numeric type".

Pragmaticly, it was always understood that fixed point multiplication (at least where the deltas are powers of two) is essentially an "integer" multiplication producing a double-length product which is then appropriately scaled to the target type of the conversion (typically also a fixed point type). That is, for the unconverted result of the multiplication, a delta equal to the product of the deltas of the two operands was always sufficiently small. However, a real literal (or other real universal operand) has no defined delta as such; indeed, real literals and static universal operands generally are required to be exact according to 4.10(4). This leads to the conclusion that universal arithmetic (involving both unbounded accuracy and/or unbounded range) may be required at RUN-TIME.

The following examples will help make this concrete.

Example 1:

```
type T is delta 0.125 range -100.0 .. 100.0;
X , Y : T;
N : constant := 13#7.0#E-1; -- 7/13th
...
X := 13.0;                  -- a model number of T
...                         -- anything to stop constant propagation
Y := T(X*N);                -- Exactly 7.0
```

Because the value of X (13.0) is a model number of T, N (7/13th) is a static universal operand, and the exact result (1.0) is also a model number of T, Y must have the exact result 7.0. In general, arbitrarily accurate computation at RUN-TIME will be required to assure this result on most reasonable machines.

In the following, assume that FLOAT is the most accurate floating point type supported by an implementation.

Example 2:

```
type T is delta 2.0**(-20) range -1000.0 .. 1000.0;
N := constant := FLOAT'SMALL*(2.0**(-20));

X : T := 2.0**(-20);                    -- a model number of T
...
Y := FLOAT(X * N);                      -- exactly FLOAT'SMALL
```

Here again, the value of X (1.0/(2.0**20))is a model number of T, N is a universal operand and the exact result (FLOAT'SMALL) is a model number of FLOAT, so that Y must have this exact result as its value. In general, unbounded range will be required at RUN-TIME to assure this result.

I do not believe that it was ever contemplated or intended by either the LDT or DRs that universal arithmetic would/should be required at run-time. If true, then further work is required to specify just what is required of an implementation in such examples as the above.

****

!section 04.05.05(10) Paul N. Hilfinger 83-10-31
!version 1983
!topic Can a real literal be an operand of a fixed point multiply?
!reference AI-00020

Well over a year ago, I had an exchange of messages with Brian Wichmann on precisely the issue that Brender has raised, but I suspect that we neglected to post the results.

Assume the following declarations

```
N: constant := ...;  -- Where N is universal_real
type Source is delta d1 ...;  -- Assume that d1 is the actual delta.
type Target is delta d2 ...;  -- Assume that d2 is the actual delta.
X: Source := ...;
```

and consider the evaluation of the expression

```
Target(N*X)
```

In the case where X and N*X are model numbers, this must yield an exact result. The question is whether this requires "infinite"-precision arithmetic at runtime. I claim it does not.

Assume that X is internally represented by MX, where

```
X = MX * d1
```

Then the problem is to find MR such that

```
MR*d2 = N * X = N * MX * d1
```

in such a way that when MR is integral, it is computed exactly. We have

```
MR = MX * (N * d1/d2).
```

The quantities N, d1, and d2 are all static. Hence, we can compute in advance integers P and Q such that

```
P/Q = N * d1/d2
```

and P/Q is in lowest terms. If MR is integral, then

```
MR = (P*MX) div Q
```

and no runtime infinite precision is required.

Of course this blithe formulation hides a host of pitfalls for the user, one of which is that an innocent-looking multiplication causes a division as well (since Q is not likely to be a power of 2 unless the programmer was more careful than he should have to be.) Another problem is that it is quite easy to get into situations where P and Q are too large, even though N may look innocent enough.

We may relieve this situation a bit with a little work. Define

```
I(x) = { ceil(x), floor(x) }
```

When computing N*X, the rules for fixed point always require only that we compute an MR that is in the set I(MX * N * d1/d2) (of course, this set has only one element when N*X is a model number.) Suppose that from range

information, we know that

| MX | <= U

and suppose that

P/Q = N * d1/d2 + epsilon

Then

P*MX div Q = trunc(P*MX/Q) = trunc(MX * N * d1/d2 + MX*epsilon)

and a sufficient condition that this quantity be in I(MX * N * d1/d2) is that

| epsilon | < 1/U, and epsilon has the same sign as N*d1/d2.

This puts much smaller demands on the sizes of P and Q. If we choose Q to be a power of two, we can use arithmetic shifting, if we are careful to remember that arithmetic right shifting computes the floor of the quotient, not the trunc:

P * MX shiftright (log2 Q) = floor(P * MX / Q) =
floor(MX*N*d1/d2 + MX * epsilon)

A sufficient condition that this quantity be in I(MX * N * d1/d2) is

0 <= epsilon < 1/U.

****

!section 04.05.05(10) Paul N. Hilfinger 83-10-31
!version 1983
!topic Can a real literal be an operand of a fixed point multiply?
!reference AI-00020

Correction on last message: floor doesn't behave quite as nicely as I initially thought. This is what comes of doing high school algebra at 1AM.

To use right shifting (which computes the floor function), use the same conditions on epsilon as for div:

| epsilon | < 1/U, and epsilon has the same sign as N*d1/d2

and use the fact that for integers x and y, y>0,

x div y = if x>=0 then floor(x/y) else floor((x+y-1)/y) fi.

(This is an example of why Guy Steele once published a note entitled something to the effect "Arithmetic right shift considered harmful.")

****

!section 04.05.05(10) Peter Belmont 83-11-02
!version 1983
!topic Can a real literal be an operand of a fixed point multiply?
!reference AI-00020 and PNF(83-10-31) (both messages)
!reference Brender(83-10-31)

Good lord.

I did not interpret these messages to mean

"Yes, a real literal CAN be an operand..."

and must, on the contrary, interpret them to mean

"Yes, we have no bananas."
and "The gobeluns'll gitche eff ..."

Ada is not supposed to be a writable language, but a readable one (for people who like longwindedness). If the programmer cannot write

... Target(X*1.734) ...

then he will jolly well have to write

... Target(X * SomeType(1.734)) ...

and thereby specify what he means. We do NOT want to require exact arithmetic of arbitrary precision at run-time. Accordingly, I recommend that the fixed-point multiply not be overloaded on universal-real. Compilers can give very useful error messages:

```
... Target( X * 1.734 ) ...
                ^
*** Error: This expression must be explicitly converted to some
***             fixed point type (LRM 4.5.5(?))
```

just as they would do if the 'Target()' had been omitted.

----- By the way,
what would we like to happen if the user has defined
a function "*"(x:fixedtype; y:float)
and writes X * 1.734
or Target(X * 1.734) ?
If real literals cannot be implicvitly converted to universal fixed, then there is some hope that his intent will be realized.

****

!section 04.05.05 (10) Ron Brender 83-11-18 83-00207
!version 1983
!topic Can a real literal be an operand of a fixed point multiply?
!reference AI-00020, 83-00128

Regarding Hilfinger's analysis of 31 Oct 83, the following is of interest. In short, it appears the approach suggested can work when the target type is itself a fixed point type, but it appears to break down when the target type is a floating point type. The argument is presented in the following:

From: BRETT 8-NOV-1983 08:48
To: BRENDER,MITCHELL,STOCKS,GROVE

Subj: Sigh

Paul Hilfinger's response is correct as far as it goes...

(1) Using the technique he gives is going to require 2N-bit integer arithmetic to implement N-bit fixed point multiplication that yields another fixed point number. This is true for other reasons as well, so isn't too much of a worry (although annoying).

(2) His mid-night high-school math did not address the issue of fixed point division yielding a fixed point result (of the form N/X, the other way round of course can be replaced by X/N <=> X*(1/N)).

In this case, and assuming D1/D2 = 1,...

```
   0  <=  P/X - N/X   <   1,      where P is an approximation for N
=>
   0  <=  P - N       <   X
=>
   N  <=  P           <   X + N
```

Fortunately, the largest N for which N/X will not overflow is only X*X, which still only requires 2*F'mantissa bits, so his technique is still adequate.

(3) His mid-night high-school math did not address the issue of conversions to FLOATING POINT types.

Consider the equality A = B * (A/B)

Let
F be the floating point type used
N = A/B
P = an approximation for N
E = P-N
X = 1.0 - greatest number for F that is less than 1.0
Y = least number for F that is greater than 1.0 - 1.0

(notice that this means X = Y/2 on most machines)

Then

```
A * (1.0 - X/2)    <   B * P      <   A * (1.0 + Y/2)
```

to guarantee that after rounding the answer is A

```
=>
   A * (1.0 - X/2)    <   B * (N+E)   <   A * (1.0 + X)
   A * (1.0 - X/2)    <   A + BE      <   A * (1.0 + X)
   - A * X/2          <       BE      <   A * X
   - A/B * X/2        <        E      <   A/B * X
```

Now, when 1/2 < A/B < 2/3, the range covered by

```
A/B - A/B*X/2   ..  A/B + A/B*X
```

is less than X in width, and thus MAY HAVE NO MODEL NUMBERS in it.

Furthermore if N is done via a divide and multiply, more precision/range than is provided by F is going to be required, which will be difficult if F is the most precise/greatest range type available in the implementation.

For instance, on the VAX-11 architecture using H-precision arithmetic, there is no H-precision number adequate to express 7.0/12.0, sigh.

(As a side-note, Goodenough has also pointed out to me that the approach can't be used in the case of certain attributes that yeild "run-time universal" values... I leave to him to present the details.)

****

!section 04.05.05 (10) J. Goodenough 83-11-18 83-00208
!version 1983
!topic real literals for fixed point multiply and divide
!reference AI-00020 83-00128

My feeling is that we should stick to the wording of the RM here. Paul Hilfinger's analysis assumes that all universal real values are static, and this is not the case. There are several ways to get arbitrary non-static universal real values, e.g., 2.0**N/3.0**M or 1.0*A'LENGTH or 1.0*T'POS(M). Since we can't interpret the RM to allow just static universal real values, the run-time consequences of using non-static universal real operands for fixed point multiplication and division are just too unpleasant to contemplate.

Even if there were not non-static universal real values I would still argue now that static universal real operands should not be allowed. Although the technique Paul sketches is probably feasible (I haven't really analyzed it

closely), it is a technique that an implementation must, in general, only support when it allows non-powers of 2 for 'SMALL. If an implementation has chosen to restrict representation clauses for 'SMALL, it should not have to do the extra work required by Paul's technique.

The current wrding of the RM certainly disallows real literals as operands in fixed point multiplication or division because there are always two fixed point types in scope (DURATION and the anonymous fixed point type required by 3.5.9(7)). Therefore, there is never a unique fixed point type that can serve as the target of an implicit conversion, and so C(1.1*F) is always ambiguous, and hence, illegal. I think we should stick with this reading of the RM.

****

!section 04.05.05 (10) P. N. Hilfinger 83-11-19 83-00217
!version 1983
!topic real literals for fixed point multiply and divide
!reference 83-00207, 83-00208, 83-00128, AI-00020

I had only intended to recapitulate what I thought was the original intent of the LDT (or at least of Brian Wichmann) on this subject. If this was not the intent, then I concur that we may as well get rid (or stay rid) of the capability multiplying universal_real*fixed_type (I would just as soon get rid of built-in fixed point types altogether anyway.) In case Brian should come up with a strong argument for wanting to provide the capability, here are a few comments.

1) "Paul Hilfinger's analysis assumes that all universal real values are static, and this is not the case. There are several ways to get arbitrary non-static universal real values, e.g., 2.0**N/3.0**M or 1.0*A'LENGTH or 1.0*T'POS(M). Since we can't interpret the RM to allow just static universal real values, the run-time consequences of using non-static universal real operands for fixed point multiplication and division are just too unpleasant to contemplate." [Goodenough]

Comment: True. In an expression such as

```
F((2.7**N) * X)    -- N an INTEGER, X of some fixed type,
                   -- F a fixed point type.
```

we would not know in advance what the proper delta is to ascribe to the left operand. The only thing that keeps us from having a simple implementation, however, is 4.5(7), which says that we are not allowed to raise NUMERIC_ERROR if the mathematical result of an operation is in a safe interval (i.e., 4.5(7) disallows hidden intermediate computations that could overflow.) Without this requirement, the computation above can be performed as follows for a delta that is a power of 2 and a machine that represents a fixed-point number, X, as a simple integer REP(X):

```
if abs REP(X) > MAX_FLOAT_MANTISSA then raise NUMERIC_ERROR;
```

```
else
   TEMP := (2.7**N) * BIG_FLOAT(X);
   if abs TEMP > MAX_ACCURATE_FLOAT_FOR_F then
      raise NUMERIC_ERROR;
   else RESULT := F(TEMP);
   end if;
end if;
```

Here, BIG_FLOAT is the highest precision floating type on the machine (mentioned in 4.10(4)); MAX_FLOAT_MANTISSA is the largest integer that can be exactly represented as a BIG_FLOAT; and MAX_ACCURATE_FLOAT_FOR_F is the maximum floating point number that can be converted accurately to an F. It may seem strange to be using floating point for this computation, but note that run-time floating point (or better) is required in any case for computations of non-static universal_real quantities (see 4.10(4)). I do not believe, in other words, that these run-time consequences are "too unpleasant to comtemplate."

2) "It is a technique that an implementation must, in general, only support when it allows non-powers of 2 for 'SMALL. If an implementation has chosen to restrict representation clauses for 'SMALL, it should not have to do the extra work required by Paul's technique." [Goodenough]

Comment: This is true if you disallow universal*fixed computations. When they are allowed, Brender's original objection seems to apply regardless of the legal values of 'SMALL. Namely, the result of a fixed point multiplication has infinite accuracy, and when the mathematically exact answer is a model number of the target type, it must be produced exactly.

By the way, support for 'SMALLs other than a power of 2 causes some interesting headaches. For example, again because of 4.5(7), a computation such as f(p*q) or f(p/q) (f a fixed type, p and q fixed variables) must not overflow if the mathematical result is in a safe interval of f. However, for general 'SMALLs, these computations will actually be translated, respectively as something like (REP(p)*REP(q)*K1)/K2, and Rep(p)*K1/(K2*Rep(q)) (or possibly Rep(p)*K1/K2/Rep(q)). What's interesting here is that if K1 is allowed not to be a power of 2, then the first computation will involve multiplication of double-length result by a single length result (rather than the usual single times single yielding double). Furthermore, if K2 is allowed not to be a power of two then the second computation will involve either division of a double length quantity by a double length quantity or division of a double length quantity by a single length quantity yielding a double length result (rather than the usual double by single yielding single).

One wants to conclude that support for 'SMALLs other than powers of two will be rare. However, this puts a slight burden on the implementor of DURATION, since machines that yield clock or timer values in units of 10E-6 seconds or 1/60 second sort of cry out for wierd 'SMALL values.

3) "His midnight high-school math did not address the issue of conversions to

FLOATING POINT types." [Brett]

Comment: True, and this is a problem. As for point (1) above, what prevents an easy solution are the stringent requirements on NUMERIC_ERROR. Specifically, in a computation such as

```
F(G * X)
```

where G is a universal_real quantity and X a fixed point variable, there are constants MAX1 and MAX2 such that we can compute

```
if abs X > MAX1 then raise NUMERIC_ERROR;
else
   TEMP := BIG_FLOAT(G) * BIG_FLOAT(X);
   if abs TEMP > MAX2 then raise NUMERIC_ERROR;
   else RESULT := F(TEMP);
   end if;
end if;
```

(The constants MAX1 and MAX2 can be improved if conversion to F rounds.)

SUMMARY

In short, it is very difficult to generate code that computes the correct answers for the cases above if we have to produce answers in all cases required by the LRM. Should it be deemed desirable to allow multiplications of universal_real by fixed quantites, it seems that a slight relaxation of 4.5(7), together with the analyses presented before (which are not particularly burdensome on an compiler implementation that has rational arithmetic.)

On the other hand, it is merely a sense of intellectual fair play that prompts me to defend a feature such as fixed point, which might just as well be deep-sixed and replaced with a specialized generic package for all I care.

****

!section 04.05.05 (10) R P Wehrum, Siemens A.G., Muenchen 83-06-02 83-00248
!version 1983
!topic Ambiguous Expressions Involving Universal Real Values

Let

```
... V1 : SOME_FIXED_POINT_TYPE := ...;
V1 := SOME_FIXED_POINT_TYPE(V1 * 3.14);
...
```

The r-h-s of the assignment should be legal (at least the programmer will expect that). However, according to the RM it is not. The type of the literal is universal_real. Thus an implicit conversion of the literal to some fixed_point type is needed (or another predefined operator for "*");

but the context does not suffice to determine the target type of the conversion; the expression is ambiguous; some semantic rule is missing.

(Cf. Section 4.5.5(10), 4.6(15).)

Is this an oversight of the language designers?

****

!section 04.05.05 (10) Terry Froggatt 86-12-10 83-00889
!version 1983
!topic Fixed Multiplication & Division with Real Literals

In Ada, a floating point number can be multiplied or divided by a real literal constant (or any universal real named number) whereas a fixed-point number cannot be: the programmer has to give the literal a type.

Real literals with fixed point multiplication 87-02-10 AI-00262/01 12

This is AI-20. In fact this restriction is unnecessary.

To perform A := A_TYPE(C*B) or A := A_TYPE(B*C) or A := A_TYPE(B/C), where C is a constant, we simply multiply or divide the scaling factor associated with the conversion of A to B, by the value of C, in the compiler; then generate exactly the same code that we would have used for A := A_TYPE(B) but using the revised scaling factor (which can now be negative). (But note that A := A_TYPE(C/B) is a different problem).

This is implemented using a multiplication by one constant then a division by another constant: the ratio of the constants being a continued fraction approximation to the scaling factor. Of all the operations on fixed point numbers which involve the use of scaling factors, this one (fixed-to-fixed conversion) is the only one which can be implemented easily.

So it is strange that the reason given for the lack of the literal operations is the uncertainty over the accuracy to which the constant has to be held at run-time, (see Ada Letters IV-2.68 & VI.6-77). There are considerably worse problems over the representation of scale factors for fixed-to-integer, fixed-to/from-float, and universal-fixed-to-any-numeric-type.

Note that it is already possible in Ada to multiply or divide fixed values by named numbers, without having to specify any reduction in the named number's accuracy:

```
PI: constant := 3.14.....;
type PI_TYPE is delta PI range 0..2*PI;
for PI_TYPE'SMALL use PI;
```

```
TYPED_PI: constant PI_TYPE := PI; -- Still as exact as the named number.
....
FIXED_VALUE := FIXED_TYPE ( FIXED_VALUE * TYPED_PI );
```

Proposed extension of the USE clause - Record 84-08-27 AI-00274/00 1

!standard 08.04 (01) 84-08-27 AI-00274/00
!class study 84-08-27 (provisional classification)
!status received 84-08-27
!topic Proposed extension of the USE clause - Record component visibility

!summary 84-08-27

!question 84-08-27

!recommendation 84-08-27

!wording 84-08-27

!discussion 84-08-27

!appendix 84-08-27

****

!section 08.04 (01) A W Thompson/GA Technologies 84-08- 83-00409
!version 1983
!topic Proposed extension of the USE clause - Record component visibility

I strongly urge that an extension to the Ada USE clause be added to the Ada standard definition. The extension I propose is to allow a statement that is analagous to the UCSD Pascal WITH statement, so that one may make record components directly visible in the same manner as one would currently use the USE clause to make package declarations directly visible. It is rather painful to have to write out a long record name (or even a hierarchy of record names) in order to access a component of a record. IN the current definition of Ada, one must write:

```
if column_dat( column (element) ).status = unassigned then
   column_dat( column (element) ).status := assigned;
   column_dat( column (element) ).assigned_elem := element;
   column_dat( column (element) ).assigned_row := row;
   column_dat( column (element) ).last_ref := current_row;
end if;
```

It would be much less tedious (and perhaps execute faster?) if one could write:

```
WITH column_dat( column (element) ) DO
   begin
      if status = unassigned then
      status := assigned;
         assigned_elem := element;
```

```
            aassigned_row := row;
            last_ref := current_row;
          end if;
        end;
```

as in UCSD Pascal. In order to avoid confusion with the current WITH statement in Ada, perhaps the word USING would be more appropriate. The syntax for the proposed extension could look something like:

```
using_statement ::= USING record_name DO
                      sequence_of_statements
                    END USING;
```

I would also propose that an extension of the USE clause be allowed in the declarative part of subprograms and block statements with the same effect as discussed above so that one could have:

```
DECLARE
  USE record_name;
BEGIN
  sequence_of_statements
END;
```

In my opinion the addition to the language of this extension would lessen the tedium of typing record names over and over and is in the same spirit as the existing USE clause. I respectfully request your consideration of this proposal.

pragma OPTIMIZE and package declarations 84-08-27 AI-00280/00 1

!standard B (08) 84-08-27 AI-00280/00
!class study 84-08-27 (provisional classification)
!status received 84-08-27
!topic pragma OPTIMIZE and package declarations

!summary 84-08-27

!question 84-08-27

!recommendation 84-08-27

!wording 84-08-27

!discussion 84-08-27

!appendix 84-08-27

****

!section B (08) Peter Belmont 84-08-16 83-00410
!version 1983
!topic pragma OPTIMIZE and package declarations

There appears to be no way to associate the pragma OPTIMIZE with a library unit package declaration since the pragma must appear within a declarative part (and since a package declaration does not contain a declaratie part).

This removes needed functionality from Ada in my opinion.

The declarations in a package declaration must be allocated and optimization strategy may determine how they are allocated. In particular, th' 'ayout of types (particularly of discriminated and variant records) must be determined; and this layout may depend (in some implementations, especially those attempting elaborate optimizations) on the user's request, most reasonably uttered via use of the praggma OPTIMIZE.

Please confirm that the LRM does not allow specification of pragma OPTIMIZE for a library unit package declaration.

Please comment on the possibility of altering Ada to allow it, for the reasons given in the discussion or otherwise.

Accuracy of Attributes of Generic Formal Types 84-10-01 AI-00285/00 1

!standard 04.10 (04) 84-10-01 AI-00285/00
!class study 84-10-01 (provisional classification)
!status received 84-10-01
!topic Accuracy of Attributes of Generic Formal Types

!summary 84-10-01

!question 84-10-01

!recommendation 84-10-01

!wording 84-10-01

!discussion 84-10-01

!appendix 84-10-01

****

!section 04.10 (04) Software Leverage, Inc. 84-01-30 83-00427
!version 1983
!topic Accuracy of Attributes of Generic Formal Types

Since attributes of generic formal types aren't static, the accuracy required for their evaluation is given by 4.10(4): "The accuracy of the evaluation of a universal expression of type universal_real is at least as good as that of the most accurate predefined floating point type supported by the implementation, apart from universal_real itself."

This makes it very difficult to portably use generic fixed point types. To illustrate this, assume we wished to completely sidestep issues of accuracy by converting from the fixed point representation to integer values:

```
type LONGEST_INTEGER is SYSTEM.MIN_INT..SYSTEM.MAX_INT;
-- Assume that, for any fixed point subtype T,
--   SYSTEM.MIN_INT*T'SMALL <= T'BASE'FIRST
--     and T'BASE'LAST <= SYSTEM.MAX_INT*T'SMALL.
...
function CONVERT(X: T) return LONGEST_INTEGER is
begin
  return LONGEST_INTEGER(X/(T'FIRST*INTEGER'(0) + T'SMALL));
  -- X/T'SMALL is ambiguous, because T'SMALL must be converted to
  -- some fixed point type, and which one isn't determined.
  -- X/T'(T'SMALL) isn't ambiguous, but will raise CONSTRAINT_ERROR
  -- if the range of T doesn't include T'SMALL.
  -- The above is a trick to arrange that only conversions to T'BASE
```

```
    -- are done;  fortunately the model number interval of the left
    -- summand is just 0.0.
  end CONVERT;
```

(That such contortions are needed is itself a defect in the language, but this is left as a topic for future comments.)

This doesn't necessarily accomplish what is desired because (e.g., if T has a length clause for T'SMALL) T'SMALL may not be a model number of the most precise floating point type. Therefore the implicit conversion of T'SMALL from universal_real to T'BASE isn't guaranteed to be accurate (it may even yield zero if T'SMALL is very small).

This problem was discovered while considering the implementation of TEXT_IO.FIXED_IO in Ada.

The four attributes which are troublesome are T'SMALL, T'LARGE, T'SAFE_SMALL, and T'SAFE_LARGE for fixed point types. (There is only a problem for generic formal types.)

It isn't clear how to minimally fix Ada to remedy this problem. One might strengthen 4.10(4) to also require exact evaluation for a convertible universal operand as defined in 4.6(10) if the operand is explicitly or implicitly converted to a real type (of course, the conversion itself need not be exact).

The next version of Ada ought to resolve this problem.

SYSTEM.MAX_DIGITS Insufficient for Portability 84-10-01 AI-00291/00 1

!standard 13.07 (03) 84-10-01 AI-00291/00
!class study 84-10-01 (provisional classification)
!status received 84-10-01
!topic SYSTEM.MAX_DIGITS Insufficient for Portability

!summary 84-10-01

!question 84-10-01

!recommendation 84-10-01

!wording 84-10-01

!discussion 84-10-01

!appendix 84-10-01

****

!section 13.07 (03) Software Leverage, Inc. 84-01-30 83-00433
!version 1983
!topic SYSTEM.MAX_DIGITS Insufficient for Portability

There seems to be no way to write a generic package which works for all floating point subtypes without (1) simulating floating point "by hand", or (2) making assumptions about the particular implementation. This problem was found while considering the implementation of TEXT_IO.FLOAT_IO.

The problem is that the predefined floating point type with the greatest precision (assuming there is only one!) isn't guaranteed to also have the largest exponent range for its safe numbers. Thus declaring

```
type LONGEST_FLOAT is digits SYSTEM.MAX_DIGITS;
```

is not sufficient to yield a floating point type whose safe numbers are certain to include those of some generic formal type T.

It should be observed that T cannot itself be used as the type of objects within the generic unit, since the range of T may not include important values like 0.0. Nor can T'BASE be used as the type of declared objects, because of 3.3.3(9). Finally, a trick like

```
type T_BASE_COPY is digits T'DIGITS
  range -T'BASE'SAFE_LARGE..T'BASE'SAFE_LARGE;
```

fails because attributes of T aren't static. (The above wouldn't always be a

copy, since T'BASE'FIRST and T'BASE'LAST might not be safe numbers, but it would be good enough.)

The minimum fix to this situation seems to be to invent a named number SYSTEM.MAX_SAFE_EMAX and demand that (1) T'SAFE_EMAX never exceeds this value for any floating point type T, (2) there is some predefined floating point type which achieves both the maximum precision and the maximum exponent range for safe numbers simultaneously. (It seems likely that for most systems, the highest precision floating point type will have the largest exponent range anyway. For other systems, requiring that a few floating point types claim less exponent range than they "actually" have seems in the same Ada spirit of portability as, say, the demand that T'DIGITS guarantees both a precision and an exponent range.)

Assuming that there isn't a programming trick we have failed to find which circumvents the problems mentioned above, we recommend that the above change or something similar be put into package SYSTEM for the next version of Ada.

Instantiating with an incomplete private type 85-02-15 AI-00327/00 1

!standard 07.04.01 (04) 85-02-15 AI-00327/00
!class study 85-02-15 (provisional classification)
!status received 85-02-15
!topic Instantiating with an incomplete private type

!summary 85-02-15

!question 85-02-15

!recommendation 85-02-15

!wording 85-02-15

!discussion 85-02-15

!appendix 85-02-15

****

!section 07.04.01 (04) D Arndt, JR Green/Bell Tech Ops 84-04-18 83-00506
!version 1983
!topic Instantiating with an incomplete private type

Consider this example:

```
Package TEST_INLINE Is -- a test to confirm that a nested package can
                       -- refer to a private type that has not been
                       -- fully declared.

    Type TI is Private;       -- private type w/o full declaration

    Package LINKS is          -- the inner nested package

        Function X Return TI; -- TI can be a parameter

    End LINKS;

Private

    TypeTI Is (TEST, IT);

End TEST_INLINE;

Generic
    Type DATA Is Private;
```

```
Package FRUSTRATION Is        -- the inner nested package expressed as
                              -- a generic.
    Function D Return DATA;

End FRUSTRATION;

With FRUSTRATION;
Package TEST_GENERIC Is  -- a demonstration that generics are not
                         -- always able to substitute for in-line
                         -- packages.
    Type TG is Private;

    Package LINKS Is New FRUSTRATION (DATA => TG);
            -- is illegal, but should be able to take the place of
            -- in-line package.
Private

    Type TG Is (TEST, IT);

End TEST_GENERIC;
```

As I understand it, the spirit of Ada concerning generics is that the instantiation of a generic package is interchangeable with an in-line package (ALRM 12). In practice, however, a serious restriction has been placed on the use of private types that does not allow generic instantiations to replace their in-line package counterparts. This problem is severe enough to cause me to choose between abandoning generics or giving up the security and control provided by privates. In the several thousand lines of Ada code I have written so far there are dozens of places where I need to instantiate a generic package with private types and not a single instance where I have desired to instantiate with a non-private type.

I am not sure what the solution to the problem is but there seem to be at least a couple of options. The most conservative is to lift the total ban on the use of non-fully declared private types as generic actual parameters and substitute the restrictions that apply to in-line packages (i.e., that the non-fully declared private types may be used in subprogram and deferred constant declarations within generic packages). This is actually how I would expect Ada to act, given the wording of chapter 12 of the language manual and the syntax rules for generic instantiation.

Another possibility is far more radical, yet makes a great deal of sense. It involves changing the language definition so that entities can be fully declared and made private simultaneously. Declarations might look something like this:

```
Private Type TEST Is (FOO, BAR);

Private t :  TEST;

Private Procedure TEST_IT;
```

The disadvantage to this sytem is that you might have to type the word Private more than once. The advantages are threefold:

1. Types would only have to be declared once instead of twice. This would aid readability and make documentation much simpler.

2. The problems associated with checking for full type declarations would be eliminated.

3. Private declarations could be interspersed throughout a package specification. This would be a very powerful feature, as illustrated by the following example:

```
Package FILE_SYSTEM Is -- a sample specification

  Private Type FILE_NODE Is
    Record
      filename : ...
      etc....
    End Record;

  Private working director : FILE_NODE; -- a private,
                                        -- global variable

  Function GET_FILENAME (NODE : In FILE NODE :=
                                      working directory)
    Return STRING;

End FILE_SYSTEM;
```

Under the current system where all private declarations come at the end of the specification, using a global variable as a default parameter to a non-private subprogram is impossible.

At the very least, the ALRM should be reworded to reflect the current state of reality with respect to generic instantiations. As it stands now, the BNF states that

generic_actual_parameter ::= type_mark (12.3)

and that

type_mark ::= type_name (3.3.2)

Nowhere in the section on generic instantiations is any restriction concerning private types mentioned, and in fact it is implied (if not stated outright) that generic packages are subject to the same rules for visibility as in-line packages.

look-ahead operation for TEXT_IO 85-03-05 AI-00329/00 1

!standard 14.03 (00) 85-03-05 AI-00329/00
!class study 85-03-05 (provisional classification)
!status received 85-03-05
!topic look-ahead operation for TEXT_IO

!summary 85-03-05

!question 85-03-05

!recommendation 85-03-05

!wording 85-03-05

!discussion 85-03-05

!appendix 85-03-05

****

!section 14.03 M. Levanto 85-02-13 83-00509
!version 1983
!topic look-ahead operation for TEXT_IO

The TEXT_IO of the Ada language, as specified in the Reference Manual (January 1983), requires that input is read to the first character not satisfying the required syntax, but the terminator is not read. This requirement means that there must be one character look-ahead. I can't see any technical reason to hide this look-ahead. I would like to write my own routines (such as GET_DATE or GAME.MOVE_IO.GET) as similar to the TEXT_IO routines as possible. To do that, I need similar look-ahead. One such look-ahead facility is in ELLIOT 803 ALGOL (April 1964), p.19, where one may use the Boolean function buffer to test the next input character against any given character. This would be proposal enough but I think more practical is

```
function NEXT(FILE : in FILE_TYPE) return CHARACTER;
```

Value of function is character to be read with a single-character GET if that is the next operation.

Record type with variant having no discriminan 85-06-18 AI-00345/00 1

!standard 03.07.03 (00) 85-06-18 AI-00345/00
!class study 85-06-18 (provisional classification)
!status received 85-06-18
!topic Record type with variant having no discriminants

!summary 85-06-18

!question 85-06-18

!recommendation 85-06-18

!wording 85-06-18

!discussion 85-06-18

!appendix 85-06-18

****

!section 03.07.03 (00) Japanese comments on DP8652 85-05-10 83-00559
!version 1983
!topic Record type with variant having no discriminants

Some programs want to handle a record type with a variant part having no discriminants. What about the proposal that ADA supports such record types?

For example, someone want to write a record type as follows (like Pascal).

```
type R is record
    case BOOLEAN is
        when TRUE  => R1 : INTEGER;
        when FALSE => R2 : CHARACTER;
                      R3 : BOOLEAN;
    end case;
end record;
```

(Note: the discriminant has no area on memory.)

In some cases a program want to input/output several types of variables from/to a file. For example, a user may want to handle a file containing some kinds of related tables. (This file might have been created by another language program.) If a record type with a variant part having no discriminants is supported, this problem can be solved.

****

!section 03.07.03 (00) J. Goodenough 85-06-17 83-00566
!version 1983
!topic re: record type with variant having no discriminants
!reference 83-00559

The possible need for variant records without an explicit discriminant component was certainly considered during Ada's design. It is true, however, that it is not always possible to read files created by another language program that does not conform to Ada's type conventions. This is certainly true in general, since there is no guarantee that the data structure will be laid out in the format expected by the Ada compiler.

Since most foreign files will be written in even numbers of words or bits, the most general approach is to define a data type that covers the expected range of record sizes, use this type to instantiate SEQUENTIAL or DIRECT IO, and then use UNCHECKED_CONVERSION to convert to an appropriate Ada type. Having variant records without discriminant components would only solve a small number of the possible problems that can occur when trying to read files created by another system.

Delete copy-in/copy-back for scalar and access parameters AI-00349/01 1
89-12-11 ST RE

| !standard 06.02 (06) 89-12-11 AI-00349/01
!class study 85-06-18
!status received 85-06-18
!topic Delete copy-in/copy-back for scalar and access parameters

!summary 85-06-18

!question 85-06-18

| !recommendation 85-06-18

!discussion 85-06-18

| !appendix 89-10-24

****

!section 06.02 (06) Japanese comments on DP8652 85-05-10 83-00564
!version 1983
!topic Delete copy-in/copy-back for scalar and access parameters

The Standard says:

For a scalar parameter, the above effects are achieved by copy: at the start of each call, if the mode is in or in out, the value of the actual parameter is copied into the associated formal parameter; then after normal completion of the subprogram body, if the mode is in out or out, the value of the formal parameter is copied back into the associated actual parameter. For a parameter whose type is an access type, copy-in is used for all three modes, and copy-back for the modes in out and out.

What about changing this paragraph into "Adopting copy-in and copy-back mechanism for parameter passing of a scalar or access parameter is not defined in the language."

Reason

(1) For a parameter whose type is an array and record, implementation needs not use copy-in and copy-back mechanism.

(2) Run-time efficiency is not good because of parameter copying.

| **********
|
| !section 06.02 (06) Erhard Ploedereder 89-10-24 83-01315

!version 1983
!topic Parameter Passing Mechanisms

!summary
It should be considered whether the distinction between scalar and other parameters can be dropped in the determination of the parameter passing mechanism.

!rationale
There is an unfortunate interaction between generics and the prescription of the parameter passing mechanism. Consider

```
generic
   type T is private;
procedure foo(X: in out T);
```

Depending on the instantiation, X will have to be passed by value or can be passed by reference. As a consequence, code sharing for this generic procedure becomes more difficult: two implementations need to be generated (assuming that passing large matrices by copy will not be acceptable to the Ada user).

Should, in reaction to other comments, the language be refined to allow the specification of the specific parameter passing mode on a per-parameter basis, then it should be seriously considered to allow implementations complete freedom of choice in the absence of such a user-provided prescription.

Subtype declarations as renamings 85-08-22 AI-00378/00 1

!standard 08.05 (16) 85-08-22 AI-00378/00
!class study 85-08-22 (provisional classification)
!status received 85-08-22
!topic Subtype declarations as renamings

!summary 85-08-22

!question 85-08-22

!recommendation 85-08-22

!wording 85-08-22

!discussion 85-08-22

!appendix 85-08-22

****

!section 08.05 (16) Brace Stout/Ron Brender 85-08-08 83-00604
!version 1983
!topic Subtype declarations as renamings

A user of one validated Ada compiler made an inquiry as follows:

> LRM 8.5(16) says that "A subtype can be used to achieve the effect of renaming a type...". This is not clearly defined. My question is:
>
> > Should the enumeration literals which belong to the type be directly visible when declaring a subtype of a non-directly visible enumeration type?
>
> Since the LRM does not specifically say, have the language lawyers said anything of this subject? This would be a good thing to have clearly stated in the next LRM when it comes out.

I responded as follows:

> There is certainly nothing in LRM 3.3.2 on subtypes nor in 8.3 on visibility that supports the notion that using a subtype declaration to "rename" an enumeration type also has the effect of making the enumeration literals of the type visible at the place of the subtype declaration as well. So our own conclusion is that this definitely does not occur.

The sentence you cite is in a "note". This means that it "is not part of the Standard", but merely points out consequences that may not be immediately obvious. In this case, we believe the purpose of the note is to explain that there is no need for a type renaming declaration because a normal subtype declaration is sufficient.

Your comment suggests a possible useful effect of a true type renaming that is not provided by a subtype declaration; one that we have never heard suggested before. We will pass this comment along to the Ada Language Maintenance Committee.

Unless I am way off base, this comment does not need to be processed by the Ada LMC as an inquiry. However, it should be retained as a comment/idea to be considered for a possible future revision of the Standard.

Allow generic subprogram bodies 85-08-22 AI-00382/00 1

!standard 12.01 (02) 85-08-22 AI-00382/00
!class study 85-08-22 (provisional classification)
!status received 85-08-22
!topic Allow generic subprogram bodies

!summary 85-08-22

!question 85-08-22

!recommendation 85-08-22

!wording 85-08-22

!discussion 85-08-22

!appendix 85-08-22

****

!section 12.01 (02) J. Goodenough 85-08-21 83-00613
!version 1983
!topic Allow generic subprogram bodies

People writing generic subprograms tend to write a generic formal part immediately preceding a body instead of applying the formal part just to the specification, e.g.,

```
generic
   ...
procedure GP is -- illegal
```

This error is made because non-generic subprograms can be declared directly by providing a body.

The language would be more uniform and less error-prone if generic subprogram units could also be declared by providing the body directly, making the above illegal example legal.

This small change could be accomplished by modifying the syntax of generic declaration to allow

```
generic_formal_part subprogram_body
```

Visibility of character literals. 85-09-16 AI-00390/00 1

!standard 04.02 (05) 85-09-16 AI-00390/00
!class study 85-09-16 (provisional classification)
!status received 85-09-16
!topic Visibility of character literals.

!summary 85-09-16

!question 85-09-16

!recommendation 85-09-16

!wording 85-09-16

!discussion 85-09-16

!appendix 85-09-16

****

!section 04.02 (05) MT Perkins/BDM 85-03-07 83-00643
!version 1983
!topic Visibility of character literals.

I would like to point out an area of ambiguity in the Ada language standard and suggest a related change to the standard. The ambiguity is illustrated by the Ada program shown below. I believe this program is correct according to the standard. It fails to compile on the Data General/Rolm compiler, however, producing the error shown in the program.

I have shown this listing to Data General Software Support. They maintain that the compiler is behaving according to the standard. They cite Section 4.2 Paragraph 5 of the Ada Language Reference Manual, which states that the character literals correspoinding to the characters contained within a string literal must be visible at the place of the string literal. They say that this paragraph implies that a Use Statement must be included in the program to make the character type directly visible. I believe that the renaming type declaration in line 11 of the example should suffice to make the character type in the string literal visible, and therefore the program should compile.

My preferred solution to this problem would be to remove Section 4.2 Paragraph 5 from the language standard. It makes protecting the visibility of a character data type awkward. In lieu of deleting Section 4.2 Paragraph 5, making the character type visible by renamin the type is preferable to a Use Statement, since other objects in the package remain not directly visible.

A member of my staff gave a copy of this program to Jerry Fisher at the recent SIGAda meeting in San Jose. Mr. Fisher requested that we also send AJPO a letter describing the problem. This letter is the result. Thank you.

```
procedure ptest is
   type roman_digit is ('I','V','X','L','C','D','M'); -- 3-14
   type roman is array (positive range <>) of roman_digit; -- 3-28
   ninty_six : constant roman := "XCVI"; -- 3-32
package dd is -- Data Dictionary
   type roman_digit2 is ('I','V','X','L','C','D','M');
   type roman2 is array (positive range <>) of roman_digit2;
   ninty_six2 : constant roman2 := "XCVI";  -- 3-32
end dd;
   subtype roman_digit2 is dd.roman_digit2;
   subtype roman2 is dd.roman2;
   thirty    : constant roman      := "XXX";
   thirty2   : constant roman2 := "XXX";

==>    THIRTY2 : constant ROMAN2 := "XXX";
***    ROMAN2 literal "XXX" contains 'X', which is not in type
       ROMAN_DIGIT2   (line 6).
***    ROMAN2 literal "XXX" contains 'X', which is not in type
       ROMAN_DIGIT2   (line 6).
***    ROMAN2 literal "XXX" contains 'X', which is not in type
       ROMAN_DIGIT2   (line 6).

begin
   null;
end ptest;
```

Incomplete types as formal object parameters 85-12-03 AI-00404/00 1

!standard 07.04.01 (04) 85-12-03 AI-00404/00
!class study 85-12-03 (provisional classification)
!status received 85-12-03
!topic Incomplete types as formal object parameters

!summary 85-12-03

!question 85-12-03

!recommendation 85-12-03

!wording 85-12-03

!discussion 85-12-03

!appendix 85-12-03

****

!section 07.04.01 (04) J. Goodenough 85-11-15 83-00688
!version 1983
!topic Incomplete types as formal object parameters

The current wording of this paragraph forbids the following useful form of generic unit:

```
generic
   type ELEMENT_TYPE is private;
package QUEUE_OPS is

   type QUEUE_TYPE is private;

   generic
      QUEUE : in out QUEUE_TYPE;        -- illegal
   package ITERATOR is
      function MORE_ELEMENTS_EXIST return BOOLEAN;
      function NEXT return ELEMENT_TYPE;
   end ITERATOR;
    ...

end QUEUE_OPS;
```

This useful form for providing an iterator over an abstract type is not allowed because of the use of QUEUE_TYPE before its full declaration. Removal of this restriction should be considered.

Allow 256 values for type CHARACTER AI-00420/03 1
88-11-08 ST RE

| !standard 02.05 (01) 88-11-08 AI-00420/03
!standard 02.01 (01)
!standard 03.05.02 (01)
!standard 04.02 (05)
!standard C (13)
!class study 85-06-18
!status received 85-06-18
!topic Allow 256 values for type CHARACTER

!summary 86-05-05

!question 86-05-05

In order to allow non-English characters in string literals, the predefined type CHARACTER should be extended to an eight-bit character set.

!recommendation 86-05-05

!discussion 86-05-05

| !appendix 88-10-20

****

!section 02.01 (01) Japanese comments on DP8652 85-05-10 83-00557
!version 1983
!topic Extend character code to allow non-English characters

The Standard says, "Each graphic character corresponds to a unique code of the ISO seven-bit coded character set (ISO standard 646)." This specification reduces applicability for processing non-English characters (for example, Japanese Kana, Korean characters, Arabic characters). We propose a extended specification of character set as follows:

The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO eight-bit coded character set (ISO standard 4873), and represented (visually) by a graphic symbol.

The characters of columns 00 to 07 of the ISO eight-bit coded character table are categorized by the same rule as the present Standard. The defined graphic characters included by columns 10 to 15 are categorized in "(f) other special characters". The other characters are not allowed in the text of a program.

The predefined type CHARACTER is a character type whose values are 256 characters of the ISO eight-bit coded character set.

The reason for this recommendation is to enforce the applicability for processing non-English characters.

****

!section 02.01 (01) R. Leavitt 86-11-24 83-00868
!version 1983
!topic French characters in strings

Canada requires its command and control programs to interact with the operator in both English and French. This appears to be difficult in Ada. Ada strings cannot contain all the characters required for French text. How can portable programs be written that prompt the operator in any language other than English?

****

!section 02.01 (00) Japanese Member Body/ISO 88-02-01 83-01014
!version 1983
!topic Requirements for character handling in programming languages

!reference ISO/TC97/SC22/N460

BASIC REQUIREMENTS

1. Every programming language standard shall have optional facilities both to handle multi-octet characters as data and to allow multi-octet characters in comments.

2. In addition to the above it is highly desirable that programming language standards have optional facilities to allow multi-octet characters in identifiers.

JAPANESE CHARACTER SET

Japanese character set is composed of Kanji, Hiragana and Katakana characters. JIS (Japanese Industrial Standard) X0208--Code of the Japanese Graphic Character Set for Information Interchange--has totally 6877 characters, including 6353 Kanji, 189 Hiragana and Katakana, 147 special symbols (including most of the ASCII special symbols), and 62 alphanumeric characters. Each character in JIS X0208 code set is represented in two-byte code.

Based on this code set for information interchange, there are several Japanese code sets for internal processing.

DATA TYPES

We need at least two character types, one (Type I) for the ISO 646 (or its

akin like ASCII) character set, and the other (Type II) for a national character set. Both types shall be usable in one program unit. In general, a character data type shall be characterized by the length of memory space occupied by a character, a single-byte for a Type I character and a multi-byte (or a single-byte) for a Type II character. In our case, the Type II character set is the Japanese character set and a Type II character occupies a two-byte memory space.

It may be possible to provide only one character type, Type II, in some programming languages if the Type II character set includes the Type I character set. In such a case, it may also be possible to use single-byte codes for the Type I characters as optimized representations.

In some programming languages where the major concern is flexibility rather than efficiency, it may be desirable to characterize the type not by memory spaces but by character sets only.

CHARACTERS IN DATA

The following facilities are required:

As internal data

1. The character type and the string (or array of character) type for both Type I and Type II are provided.

2. Type I data (single-byte codes) and Type II data (multi-byte codes) are assumed not to be mixed within a character type variable.

3. Data length is counted by the number of characters not by the number of bytes.

4. It is desirable that the same set of standard functions for character data are provided, using the same function name as far as possible, for both Type I and Type II.

5. Conversion functions for common characters between Type I data and Type II data are provided, if both types have considerable number of common characters, as in our case.

As data in program text

Two forms of character literals for Type I and Type II are provided.

Examples: "East West South North" -- Type I
N" (japanese symbols)" -- Type II

As external data

1. Mixture of Type I and II characters is allowed in external files. In general, it is desirable that only the input/output procedures handle

this mixture. Therefore, it is desirable, for example, that the input procedures assign their values separately to appropriate (Type I or II) internal variables according to the format specifications in the input statements. However, in some programming languages, especially in languages for system programs, it may be required that some variables have mixed type strings. The type of such a variable may be neither Type I nor Type II, but type or integer.

2. Usually, the Kanji characters are twice as wide as alphanumeric characters in printed form and we have two widths (normal-width and double-width) of alphanumeric characters in Japanese word processors. However, in general, the character width shall have nothing to do with the type of characters in output formatting. The character width shall be specified by some kind of format specification or by default, for example, normal-width for the Type I and double-width for the Type II except Katakana.

Comments

The type II character shall be allowed in comments.

Identifiers

It is desirable that the Type II characters are allowed in identifiers.

Keywords

Keywords shall be composed only of the Type I characters.

| **********

| !section 02.01 (00) David B. Kinder, BiiN 88-10-20 83-01026
| !version 1983
| !topic 8-bit character type

| AI-420 (submitted over 3 years ago) brought up the issue of an an 8-bit character type. This is now becoming a hot issue with us and our Non-US customers who are insisting on support for their (non-English) characters in STRING variables and in TEXT_IO. I've seen packages that define a new extended character type, however this character type can't be used with TEXT_IO, so a whole new (and non-standard) I/O package must be used.

| Have there been any further thoughts by the LMC/ARG on supporting 8-bit characters? Can we get some ruling on AI-420? Is ISO 4873 being considered?

| -- David Kinder, BiiN

Eliminate pragma ELABORATE 86-05-05 AI-00421/00 1

!standard 10.05 (04) 86-05-05 AI-00421/00
!class study 86-05-05 (provisional classification)
!status received 86-05-05
!topic Eliminate pragma ELABORATE

!summary 86-05-05

!question 86-05-05

!recommendation 86-05-05

!wording 86-05-05

!discussion 86-05-05

!appendix 85-07-21

****

!section 10.05 (04) Paul N. Hilfinger 85-06-07 83-00553
!version 1983
!topic Usability of pragma ELABORATE

PROBLEM

A number of recent messages to ada-info have pointed up a possible design flaw in the semantics of library unit elaboration. While there is no semantic inconsistency, the problem is sufficiently serious that some action may be called for.

Consider the following library package body.

```
with TEXT_IO; use TEXT_IO;
package body IO_UTILITIES is
    package MY_FLOAT_P is new FLOAT_IO(FLOAT);
    ...
end IO_UTILITIES;
```

At first glance, this is a completely harmless-looking package. Unfortunately, it is wrong. It is completely proper for an implementation to raise PROGRAM_ERROR upon elaborating the generic instantiation of FLOAT_IO. This is because nothing in the program assures that the body of TEXT_IO (and hence, the body of TEXT_IO.FLOAT_IO) will be elaborated before the body of IO_UTILITIES.

Now you might think that this is easily fixed with pragma ELABORATE:

```
with TEXT_IO; use TEXT_IO;
pragma ELABORATE(TEXT_IO);
package body IO_UTILITIES is
    package MY_FLOAT_P is new FLOAT_IO(FLOAT);
    ...
end IO_UTILITIES;
```

Unfortunately it NEED NOT be. Nothing in the Standard appears to prevent the implementation from still raising PROGRAM_ERROR as a result of the instantiation of FLOAT_IO. To see how this could happen, suppose that the package TEXT_IO is implemented entirely in Ada, and that its bodies are ordinary Ada package bodies with the following contents:

```
with ..., SUPPORT_PACKAGE, ...; ...
package body TEXT_IO is
    ...
    package body FLOAT_IO is
        ...
        A_CONSTANT: constant ... := SUPPORT_PACKAGE.F(...);
        ...
    end FLOAT_IO;
    ...
end TEXT_IO;
```

Note that the initialization of TEXT_IO itself need not require SUPPORT_PACKAGE. Hence, TEXT_IO itself may not need a pragma ELABORATE for SUPPORT_PACKAGE; there is not, in general, a need to elaborate the body of SUPPORT_PACKAGE before that of TEXT_IO.

The problem, of course, is that programmers are not supposed to have to know about the packages used to implement ones that they use. This is particularly serious with standard library packages, where it now appears that there is no way to make the definition of the package IO_UTILITIES entirely portable, even if it uses nothing but features of TEXT_IO that are completely specified in chapter 14. However, I also think that it can be a serious problem for user-defined packages as well. In general, whenever one needs a pragma ELABORATE for library unit A, one also may need a pragma ELABORATE for each of the library units used in the implementation of A, and yet one is not supposed to have to know about the latter units.

I'll go even farther. It is annoying (and I think many will find it a surprising source of error) that one should even have to put in an ELABORATE pragma for TEXT_IO at all. After all, it IS a standard package, mandated by the language, and we are used to having such facilities "always there" without our having to think about them.

QUESTION

Should implementors be allowed to raise PROGRAM_ERROR in cases such as illustrated above (i.e., involving language-prescribed standard

packages)? If not, on what basis do we disallow it?

COMMENT

Based on this discussion, it's become clear to me that implementors should think about coming up with cleverer ways of scheduling library unit elaboration order. It is likely to become an important usability issue. The original design of the language required that the bodies be elaborated in such an order that access before elaboration did not occur. Unfortunately, this in general requires solving the Halting Problem. Attempts to formulate effective, sufficient procedures (which would disallow some, but not too many, working programs) kept leading to rather complex rules, and pragma ELABORATE was adopted as a compromise. The examples here suggest that we need to re-visit the issue.

****

!section 10.05 (04) Peter Belmont 85-07-15 83-00586
!version 1983
!topic Usability of pragma ELABORATE
!reference 83-00553

I wish to agree with Paul Hilfinger's note of 85-6-7 (83-00553) that something needs to be done about pragma ELABORATE. Let me add examples and arguments that relate to it.

Pragma ELABORATE treats two kinds of problems. First is the knowledge that serious use of a package requires its prior initialization through its body's elaboration. Pragma elaborate is ill-suited to solving this problem. I suggest a self-referential use of pragma elaborate to help with this problem.

Second is the knowledge that serious use will be made of a package before subprogram "main" has been called. This is the problem that Paul recognized, a problem of cascading. I treat these problems separately.

------- package implementer's knowledge ------

There are three "kinds" of initialization of a package. First is the explicit initialization which occurs in the package specification's elaboration. Second is the initialization which occurs from elaboration of the package body. Third is initialization provided by calls to initialization routines explicitly provided in the package.

The first offers no problems, nor any ways out of other difficulties. A package SPEC is always elaborated at the right time.

The third is not useful except as there is knowledge of the package's initialization requirements on the part of (the coders of) other code; in this case, it is sufficient to all needs, obviating the need

for pragma ELABORATE. Thus

```
package UTIL is
  function INIT return boolean;
  ...
end UTIL;
----------
package USER is
  x : boolean := UTIL.INIT;
  ...
end USER;
```

It is my contention that this (admittedly inelegant) coding style allows all that pragma ELABORATE allows. Indeed, it allows more, for one could have different kinds or levels of initialization for different kinds of users:

```
x : boolean := UTIL.INIT_1;
y : boolean := UTIL.INIT_2;
```

A library-unit procedure can always be coded within a library-unit package to achieve the functionality of the procedure with pragma ELABORATE:

```
with UTIL;
package PROC_PACK
  x : boolean := UTIL.INIT;
  procedure PROC ( arg : UTIL.T1 );
end PROC_PACK;
```

Therefore, the first kind of initialization may be neglected in relation to pragma ELABORATE and the third kind of initialization is more than equivalent (functionally, not in elegance) to what pragma ELABORATE offers. The question is: how could pragma ELABORATE be given power not already available in terms of the second kind of initialization, elaboration of the package body?

I'd suggest that all that needs being done is to give another meaning to pragma ELABORATE: when used in relation to the package BODY (i.e., self-referentially), it would mean:

> regard the package spec and its body as unitary at elaboration time, so that the body must be elaborated before any library unit which (directly or indirectly) WITHs the package.

This use of pragma ELABORATE allows the coder of the package, who alone knows the elaborational requirements of the implementation, to decide the question of elaboration order.

The present meaning of pragma ELABORATE makes it a tool of the user

who (thinks he) knows something about a package's implementation. Any user who has such knowledge might as well make an explicit call to achieve desired initialization.

------ package caller's knowledge ------

But Paul's note goes to a different question. No-one doubts that a package's body must (in general) be elaborated before (serious) use is made of its subprograms, etc. And no-one doubts that the package's body will be elaborated before the body of the "main" program is called. The issue is providing for the case that serious use is made of a package before "main" is called. Here, the implementer of user code has the knowledge, based on fuzzy knowledge of the utility code, to decide questions of elaboration order. The pragma ELABORATE has been provided to empower a writer of user code to say: " my code uses that package before MAIN is called."

Paul points out the need for transferability of that empowerment. I'd recommend the following.

Allow the user to write pragma SECONDARY_ELABORATE(x) meaning pass through a pragma ELABORATE to x. Thus

```
package PACK1 ...

with PACK1; pragma seconday_elaborate(PACK1);
package PACK2 ...

with PACK2; pragma secondary_elaborate(PACK2);
package PACK3 ...

...

with PACK3; pragma elaborate(PACK3);
package USER_CODE...
```

Here, the pragma elaborate with USER_CODE will fire off a cascade of implicit pragma elaborates hinted at but not demanded by the pragma secondary_elaborates. Did USER_CODE omit its pragma, there would be no implicit pragma elaborates.

Semi-constrained subtypes 86-06-19 AI-00427/00 1

!standard 03.05 (02) 86-06-19 AI-00427/00
!class study 86-06-19 (provisional classification)
!status received 86-06-19
!topic Semi-constrained subtypes

!summary 86-06-19

!question 86-06-19

!recommendation 86-06-19

!wording 86-06-19

!discussion 86-06-19

!appendix 86-06-19

****

!section 03.05 (02) MATS WEBER, DALIN SOFTWARE 86-04-28 83-00745
!version 1983
!topic Semi-constrained subtypes

ONE POINT IS THAT OF THE INTEGER SUBTYPES NATURAL AND POSITIVE WHICH ARE UNFORTUNATELY CONSTRAINED AND ONE MAY WRITE :

```
type TEXT (LENGTH : NATURAL) is

  record
    ST : STRING (1..LENGTH)
  end record

T : TEXT;
```

WHICH IS CATASTROPHIC BECAUSE THE OBJECT T REQUIRES INTEGER'LAST BYTES OF MEMORY AND IS CONSTRAINED. IN SUCH A CASE IT SHOULD NOT BE ALLOWED TO DECLARE AN OBJECT OF TYPE TEXT WITHOUT EXPLICITLY CONSTRAINING IT. ONE WAY TO ACHIEVE THIS WOULD BE TO INTRODUCE "SEMI-CONSTRAINED SUBTYPES" WHICH ARE CONSTRAINED ONLY WITH ONE (UPPER OR LOWER) BOUND. THE SUBTYPES NATURAL AND POSITIVE WOULD BE DECLARED AS FOLLOWS, SPECIFYING ONLY THE LOWER BOUNDS :

```
subtype NATURAL  is INTEGER range 0.. ;
subtype POSITIVE is INTEGER RANTE 1.. ;
```

AND THE ATTRIBUTES NATURAL'CONSTRAINED AND POSITIVE'CONSTRAINED WOULD

YIELD THE VALUE FALSE.

Allow array type definition for record compone 86-06-19 AI-00429/00 1

!standard 03.07 (02) 86-06-19 AI-00429/00
!class study 86-06-19 (provisional classification)
!status received 86-06-19
!topic Allow array type definition for record component

!summary 86-06-19

!question 86-06-19

!recommendation 86-06-19

!wording 86-06-19

!discussion 86-06-19

!appendix 86-06-19

****

!section 03.07 (02) MATS WEBER, DALIN SOFTWARE 86-04-28 83-00744
!version 1983
!topic Allow array type definition for record component

THE 1981 ADA LANGUAGE DEFINITION ALLOWED AN ARRAY TYPE DEFINITION FOR THE TYPE INDICATION OF A RECORD COMPONENT, BUT THIS WAS REMOVED IN THE 1983 STANDARD. I THINK THIS MODIFICATION INTRODUCES SOME DISADVANTAGES, FOR EXAMPLE IF ONE NEEDS TO REPRESENT A PERMUTATION OF THE INTEGERS RANGING FROM 1 TO N, THEN THE FOLLOWING TYPE DEFINITION WOULD BE NATURAL:

```
type PERMUTATION (N : NATURAL) is
  record
    PERM : array (1..N) OF INTEGER range 1..N;
  end record;
```

BUT SINCE THE 1983 STANDARD THIS IS NOT POSSIBLE ANYMORE AS A RECORD COMPONENT DEFINITION DOES NOT ALLOW A CONSTRAINED ARRAY DEFINITION. INSTEAD ONE IS OBLIGED TO WRITE :

```
type PERM_ARRAY is array (INTEGER range <>) OF INTEGER;

type PERMUTATION (N : NATURAL) is
  record
    PERM : PERM_ARRAY (1..N);
  end record;
```

WHICH IS NOT AS GOOD BECAUSE THE COMPONENTS OF THE ARRAY CANNOT BE CONSTRAINED AND ADDITIONALLY THE ARRAY TYPE IS OF NO USE IN THE REST OF THE PROCEDURE OR PACKAGE.

THE CASE OF THE PERMUTATION IS QUITE FREQUENT IN COMBINATORIAL OPTIMIZATION AND PROBABLY IN MANY OTHER CASES IT WILL BE IMPORTANT TO

HAVE RECORD COMPONENTS THAT ARE ARRAYS WHOSE BOUNDS AND COMPONENTS DEPEND ON A DISCRIMINANT.

Time zone information in package CALENDAR 86-07-10 AI-00442/00 1

!standard 09.06 (07) 86-07-10 AI-00442/00
!class study 86-07-10 (provisional classification)
!status received 86-07-10
!topic Time zone information in package CALENDAR

!summary 86-07-10

!question 86-07-10

!recommendation 86-07-10

!wording 86-07-10

!discussion 86-07-10

!appendix 86-07-10

****

!section 09.06 (05) Software Leverage, Inc. 84-01-28 83-00296
!version 1983
!topic Usage of CALENDAR.CLOCK

What is the intended usage of CLOCK? Is it simply meant to be a value for printing on listings, as it were? Or is it intended that it be usable to determine ordering of events in, say, tasking?

If the latter, it should be stated that calls to CLOCK produce monotonically nondecreasing results. This has nontrivial consequences because, for example, CLOCK may not return Daylight Savings Time if this is required. If this is what was intended, we suggest that future versions of Ada add two declarations to CALENDAR:

```
subtype TIME_ZONE is INTEGER range -12..12;
-- 0 is GMT, 1 is 1 zone east of GMT, etc.
-- 12 and -12 are the same time zone but on opposite sides of the
-- International Date Line.
LOCAL_ZONE: constant TIME_ZONE;
-- Value is given in private part
```

and state that Standard Time for the local zone is returned. (If the program is meant to run within an airplane, there is no need for the value of LOCAL_ZONE to correspond to the physical location of the processor, so, for example, Greenwich mean time could be used.)

****

!section 05.06 (07) R.G. Cleaveland 86-06-25 83-00758
!version 1983
!topic Time zone information in package CALENDAR
I note that the package CALENDAR is not specified to represent time in any particular world time zone. Implementations historically have reflected the year, day, hour, minute and second returned by the underlying operating system clock. While the operating systems usually also report time zone information, this aspect is ignored in presenting the data to an Ada application through the subprogram CALENDAR.CLOCK.

There are a great number of applications which require the collation and comparison of time data of varying time zones, and a need to present time information to users in terms they can most easily relate to - that is, "local" time. The application I had that brings this to mind is the sorting of net messages received by time of origination, where the messages were originated on many different hosts at different time zones.

It seems to me that there are two approaches to effecting a fix. One is to add an element for time zone to the type TIME, and the other is to "clarify" the specification of TIME semantically to mean GMT, and then do zone adjustments on input/output.

a. To add an element "zone" to "time" while preserving compatability with the current definitions would require additional parameters in SPLIT and TIME_OF as well as an additional subtype for zone and a new function to extract that parameter similar to the function YEAR. Programs prepared on the basis of the current specification would likely require modification to accomodate the extra parameter in SPLIT and TIME_OF when the new package was introduced. The only way around this is to have the enhancement reflected not in CALENDAR but in a required replacement with a different name (WORLD_CALENDAR?).

b. If a clarification were issued such that "TIME" was understood to be in terms of GMT, then it would generally be up to the vendors of CALENDAR bodies to effect the standard in the implementation of the function CLOCK. They would do this by looking at the operating system to get zone information and then adjust the time from the clock to create GMT. In most cases, until that change were effected application programs would receive operating system time, usually (but not exclusively) "local". Some hosts' clocks are set to GMT. In some cases applications would require revision to accomodate the change; the program that had been printing out "this message received at .... EDT" because it (necessarily) made an assumption about zone would have to change, either by indicating the zone as GMT or converting to EDT.

AN alternative of preparing a super-package above CALENDAR which

would accomodate time zone information was not explored on the principle that CLOCK hides any time zone information available from its source, and the super-package CLOCK would have to "go around" CALENDAR.CLOCK to get the information from (probably) that same source. Hence the super-package would necessarily constitute a replacement for that one function at least.

I recommend alternative (b). In many cases the implementation could be effected by directing the hosts system clock be set to GMT, and making a minor change or two to the (not standardized) application programs which input/output time. I would be interested in other's views on the subject.

R.G. CLEAVELAND

****

!section 09.06 (07) S.F. Landherr 86-06-26 83-00760
!version 1983
!topic Time zone information in package CALENDAR
!references 83-00758

I agree that the CALENDAR package is deficient in not catering for world time zones.
However most users are interested in local time only, so any extension of CALENDAR and any interim solution should be transparent to such users.

In addition, a calendar package with time zones should :
(1) be useful all over the world (no parochialism),
(2) cater for those places on half-hour time zones (eg South Australia),
(3) have access to the current local time zone,
(5) allow an (alterable) default zone for interpretation of times,
(4) allow conversions between time zones, and
(5) allow arithmetic and comparisons between times in different zones.

To actually designate the time zones, I recommend the usual system of letters:
Z for GMT
A, B, C, D, E, F, G, H, I, K, L, M for hours ahead of GMT
N, O, P, Q, R, S, T, U, V, W, X, Y for hours behind GMT
two-letter combinations (eg IK) for half-hour zones

My suggestion for the long-term solution (i.e. for Ada 88) is:

```
package CALENDAR is
  type TIME is private;      -- includes the time zone

  subtype YEAR_NUMBER is .....
  ...                        -- unchanged
```

```
  subtype DAY_DURATION is ....

  type TIME_ZONE is (Y,YX,X,XW,W, .. ,N,NZ,Z,ZA,A,AB, .. ,L,LM,M);
      -- half hour time zones forward from International Dateline

  function LOCAL_TIME_ZONE return TIME_ZONE; -- from operating system

  DEFAULT_ZONE : TIME_ZONE := LOCAL_TIME_ZONE;

  function CLOCK return TIME;

  function YEAR    (DATE : TIME;
                    ZONE : TIME_ZONE := DEFAULT_ZONE) return YEAR_NUMBER;
  function MONTH   ...  return MONTH_NUMBER; --
  function DAY     ...  return DAY_NUMBER;   -- like YEAR
  function SECONDS ...  return DAY_DURATION; --

  procedure SPLIT (DATE    : in TIME;
                   YEAR    ...
                   MONTH   ...
                   DAY     ...
                   SECONDS ...
                   ZONE    : in TIME_ZONE := DEFAULT_ZONE)

  function TIME_OF (YEAR    ...
                    MONTH   ...
                    DAY     ...
                    SECONDS ...
                    ZONE    : in TIME_ZONE := DEFAULT_ZONE) return TIME;

  function "+"  ...
  ...               -- unchanged
  function ">=" ...

  TIME_ERROR ...  -- unchanged

private
  -- implementation dependent
end CALENDAR;
------------------------------------------------------------------------------
```

In theory, implementations would be free to use any time zone for the internal representation of TIME values, and the time zone returned by function CLOCK is immaterial, provided that the TIME value is in fact the correct world-time. However, for portability, all Ada TIME values are best stored in GMT.

Access to LOCAL_TIME_ZONE is required for portability of programs that input or output years, months, days etc.

Conversion between time zones is through the ZONE parameter. A function to extract the time zone of a TIME value is not needed, (and could be abused).

If DEFAULT_ZONE is left unchanged, then the use of time zones is quite transparent to "local-time-only" users.

For convenience, parochial acronyms for civil time zones can be introduced by additional, normal packages: eg

```
with CALENDAR; use CALENDAR;
package USA_TIME_ZONES is

  type USA_TIME_ZONE is (PST, PDT, MST, MDT, CST, CDT, EST, EDT);
                        -- order not significant

  type CIVIL_TIME_MODE is (STANDARD, DAYLIGHT);

  function USA_TIME_ZONE (WORLD_ZONE : TIME_ZONE
                          MODE : CIVIL_TIME_MODE ) return USA_TIME_ZONE;

  function PST return TIME_ZONE;  -- returns U
  function PDT return TIME_ZONE;  -- returns T
  function MST return TIME_ZONE;  -- returns T
  function MDT return TIME_ZONE;  -- returns S
  function CST return TIME_ZONE;  -- returns S
  function CDT return TIME_ZONE;  -- returns R
  function EST return TIME_ZONE;  -- returns R
  function EDT return TIME_ZONE;  -- returns Q

  ERROR : exception;  -- raised by USA_TIME_ZONE for illegal inputs

end USA_TIME_ZONES;
```

As an interim solution, I suggest that the enhanced CALENDAR package be provided as an additional pre-defined package under a different name (eg WORLD_CALENDAR) for use by those who need it.

It could be argued that package CALENDAR is also deficient in not providing facilities for:

(1) time of day (both 12 hour and 24-hour clock)
(2) names of months (abbreviated and in full)
(3) names of days of week (abbreviated and in full)
(4) correct day of week for any given date
etc

However none of these require unusual interaction with the operating system, and thus need not be part of a predefined package, but can be provided as normal packages.

Stefan F. Landherr

****

!section 09.06 (07) M. Moore 86-06-29 83-00762
!version 1983
!topic Time zone information in package CALENDAR
!references 83-00760

> From: Stefan.Landherr@sei.cmu.edu
> ...
> In addition, a calendar package with time zones should :
> (1) be useful all over the world (no parochialism),

> (2) cater for those places on half-hour time zones (eg South Australia),
> ...
> To actually designate the time zones, I recommend the usual system of letters:
> Z for GMT
> A, B, C, D, E, F, G, H, I, K, L, M for hours ahead of GMT
> N, O, P, Q, R, S, T, U, V, W, X, Y for hours behind GMT
> two-letter combinations (eg IK) for half-hour zones

This is a step in the right direction, but does not go far enough. To be useful all over the world, even half-hour time zones are not sufficient; some places differ from the common time zones by 15 or 35 minutes or other odd amounts (not to mention places in the Middle East which use local sun time, which varies from day to day!)

The last draft I saw of the Fortran 8x standard contained a simple solution: time zone information is stored as an integer number of minutes which must be subtracted from local time to yield GMT.

Martin Moore (mooremj@eglin-vax.arpa)

Should allow raising of an exception in anothe 86-08-06 AI-00450/00 1

!standard 11.03 (02) 86-08-06 AI-00450/00
!class study 86-08-06 (provisional classification)
!status received 86-08-06
!topic Should allow raising of an exception in another task.

!summary 86-08-06

!question 86-08-06

!recommendation 86-08-06

!wording 86-08-06

!discussion 86-08-06

!appendix 86-08-06

****

!section 11.03 (02) A.D. Wolfe, Jr. 86-07-29 83-00782
!version 1983
!topic Should allow raising of an exception in another task.

I was trying to write a program with two tasks, one of which would interrupt the other in the middle of less-important processing. There seems to be no way to do this without aborting the task or inserting a useless rendezvous between the two tasks. I think that a task specification should allow the definition of externally-accessible exceptions, which can be raised by other tasks; perhaps restricting the scope to a parent task. What this really amounts to is asynchronous (ie non-rendezvous) intertask communications, which seem otherwise not to be handled...

I have found no references implying that this can be handled by embedding a task within a package.

Task entries as formal parameters to generics 86-08-06 AI-00451/00 1

!standard 12.01 (02) 86-08-06 AI-00451/00
!class study 86-08-06 (provisional classification)
!status received 86-08-06
!topic Task entries as formal parameters to generics

!summary 86-08-06

!question 86-08-06

!recommendation 86-08-06

!wording 86-08-06

!discussion 86-08-06

!appendix 86-08-06

****

!section 12.01 (02) Roger Racine 86-07-02 83-00777
!version 1983
!topic Task entries as formal parameters to generics

Can anyone give a reason why entries are not allowed as parameters to generics? The ability to use selective calls can sometimes be necessary to avoid deadlocks, and making a task entry look like a procedure makes it impossible to use a selective call.

It would seem easy enough to add a "with entry _____" construct to the formal part of the generic, following the same rules as for procedures. Then any entry having the correct parameters could be used as the actual parameter in an instance of the generic.

One reason suggested by the person who asked me about it is that the syntax of an entry call always has the "TASK.ENTRY" form. Is there some problem with compiler construction or readability that would make the procedure syntax a problem? One could (if necessary) make the formal parameter have the correct syntax ("with entry T.E").

There is, of course, a way around this problem. I would appreciate comments as to whether the workaround is as distasteful to others as it is to me. One can create a procedure which has the same parameters as the entry, plus a parameter of type DURATION (mode "in") and an "out" parameter of type BOOLEAN to specify whether the rendezvous occurred. This approach works well when the entry has no "out" parameters, but forces the procedure to put something in the parameters if they exist.

By the way, it was not possible to use an access to a task type,
because the tasks were being declared in a generic; therefore, each
instance created (necessarily for this application) a new type.
The entries needed for the generic I am trying to design were identical
(START_TASK, STOP_TASK, that kind of entry) in all instances.

If anyone has a more elegant solution, I would love to look at it.

Please send them to INFO-ADA@ADA20.ISI.EDU so that
1) Others can see them.
2) Others do not duplicate them.

Roger Racine
C.S. Draper Laboratory
RRACINE@ADA20.ISI.EDU

****

!section 12.01 (02) Roger Racine 86-07-22 83-00778
!version 1983
!topic Task entries as formal parameters to generics

It would seem that I did not make my problem clear. I do not want to
have entries look like procedures; I want entries to look like entries.
Entries are callable with protections from infinite delays. Procedures
are not. Try using a

```
select
   MIGHT_BE_AN_ENTRY;
or
   delay 1.0;
end select;
```

The problem arose when the task containing the entry was not in the
correct mode to accept the call. There is no way for the generic to
wait some amount of time for the call to be accepted, and get control
back if it is not.
Sorry about the delay on this, but I went on vacation for two weeks.

Roger Racine

"generic_type_definition" should have generic 86-08-06 AI-00452/00 1

!standard 12.01 (02) 86-08-06 AI-00452/00
!class study 86-08-06 (provisional classification)
!status received 86-08-06
!topic "generic_type_definition" should have generic record types

!summary 86-08-06

!question 86-08-06

!recommendation 86-08-06

!wording 86-08-06

!discussion 86-08-06

!appendix 86-08-06

****

!section 12.01 (02) A.D. Wolfe, Jr. 86-07-29 83-00781
!version 1983
!topic "generic_type_definition" should have generic record types

I ran into some serious problems as I was trying to write a few useful generic packages. There are a number of instances where it is critical for a generic package to be able to decompose a record-type generic argument (in an iterative manner, obviously). Two truly critical problems are generic relational database access and generic network access (through an external data representation like Sun XDR.) The former problem is well attested by papers from the 1985 Paris Ada International Conference, ie those by Poutanen et al and Smith et al. These people have found it necessary to create preprocessors allowing the embedding of useful abstract relational database access mechanisms in Ada code. This approach is abhorrent and illustrates the gravity of this problem.

The areas of the definition which relate to this problem are, first, the declaration of formal generic parameters (a "record" type is needed), and second, the definition of standard attributes by which the components of the record may be (recursively) extracted. I tossed together three quick illustrations of this problem, as follows:

SOFTWARE ENGINEERING PROBLEMS:

1. Generic record-structured I/O testbench.
   GENERIC

```
        TYPE Value_for_Test IS PRIVATE;
    PACKAGE Test_Bench IS
        FUNCTION Value_from_Tester
            (Prompt: IN STRING)
            RETURN Value_for_Test;
        PROCEDURE Display (Value: IN Value_for_Test);
        FUNCTION High_Value_s RETURN Value_for_Test;
        FUNCTION Low_Value_s RETURN Value_for_Test;
        FUNCTION Random_Content_s RETURN
            Value_for_Test;
        FUNCTION Tester_Value_from_Menu RETURN
            Value_for_Test;
            -- Goes through each component
            -- and asks tester if he wants
            -- high, low, random, or his own
            -- manually-input value
    END Test_Bench;
```

2. Generic Relational DBMS Access Package.

```
    PACKAGE Relational_DBMS_Access IS

        GENERIC
            TYPE Retrieval_Tuple IS RECORD;
            Query_Statement: IN STRING;
            TYPE Parameter_Tuple IS RECORD;
        PACKAGE Retrieval IS
            End_of_Fetch: EXCEPTION;
            PROCEDURE Query
                (Criteria: IN Parameter_Tuple);
            FUNCTION Next_Tuple RETURN
                Retrieval_Tuple;
        END Retrieval;

        SQL_Syntax_Error: EXCEPTION;
        SQL_Internal_Error: EXCEPTION;

        GENERIC
            Statement: IN STRING;
            TYPE Parameter_Tuple IS PRIVATE;
        PROCEDURE Execute_SQL;

    END Relational_DBMS_Access;
```

3. Generic Record Translator to XDR External Data Representation. Must go through each component and each subrecord recursively and execute a translation routine based on whether the component is ASCII, integer, real (float or fixed), etc.

STORAGE_SIZE for Tasks 86-08-06 AI-00453/00 1

!standard 13.02 (10) 86-08-06 AI-00453/00
!class study 86-08-06 (provisional classification)
!status received 86-08-06
!topic STORAGE_SIZE for Tasks

!summary 86-08-06

!question 86-08-06

!recommendation 86-08-06

!wording 86-08-06

!discussion 86-08-06

!appendix 86-08-06

****

!section 13.02 (10) Bob Conti/Ron Brender 86-07-09 83-00776
!version 1983
!topic STORAGE_SIZE for Tasks

There is a problem associated with giving a STORAGE_SIZE representation specification for a task that has been declared as single task in a later declaration. The problem shows up when porting from one implementation to another.

To illustrate the problem, suppose that one has written a very large program, and one has declared many single tasks declared as later declarations. Suppose that the default STORAGE_SIZE used by the host implementation happens to be adequate for these tasks. Now, suppose that, much later, one tries to port to another implementation. It is highly likely that the default STORAGE_SIZE used by that implementation will be different than that used by the original implementation. If the default happens to be less, a STORAGE_SIZE representation spec might be needed for all those tasks.

Adding the STORAGE_SIZE representation spec can force a considerable rewrite of the program. This is because the rep spec is be a basic declaration and applies only to task types. We must thus split the single task declarations into task type declarations and task object declarations. Then we must move these new declarations so they are in the basic declarations. Even more revision may be necessary if the task specifications referenced other later declarations.

The following is an actual program submitted by one of our users and its revision to add the rep spec. Note how much rearrangement was required.

EXAMPLE. Program as it worked on original implementation.
STORAGE_ERROR was raised after porting because
array C exceeded the size of the default task stack.

```
with TEXT_IO ; use TEXT_IO ;
procedure BUNCH is
   COUNTER : INTEGER := 0 ;
   X : INTEGER ;

   function F2 ( PARAMETER : INTEGER ) return INTEGER is
      C : array ( 0 .. 40_000 ) of INTEGER ;
   begin
      PUT_LINE ( " Inside function F2 " ) ;
      C ( PARAMETER ) := PARAMETER ;
      C ( PARAMETER + 1 ) := C ( PARAMETER ) + 1 ;
      return C ( PARAMETER + 1 ) ;
   end F2 ;

   function F1 return INTEGER is
      B : INTEGER := F2 ( COUNTER ) ;
   begin
      PUT_LINE ( " Inside function F1 " ) ;
      COUNTER := COUNTER + 1 ;
      return COUNTER ;
   end F1 ;

   task T1 is
      entry E1 ;
   end T1 ;

   task T2 is
      entry E2 ;
   end T2 ;

   task T3 is
      entry E3 ;
   end T3 ;

   task body T1 is
      A : INTEGER := F1 ;
   begin
      PUT_LINE ( " After begin in T1" ) ;
      accept E1 ;
      PUT_LINE ( " E1 accepted " ) ;
   end T1 ;

   task body T2 is
```

```
      A : INTEGER := F1 ;
   begin
      PUT_LINE ( " After begin in T2" ) ;
      accept E2 ;
      PUT_LINE ( " E2 accepted " ) ;
   end T2 ;

   task body T3 is
      A : INTEGER := F1 ;
   begin
      PUT_LINE ( " After begin in T3" ) ;
      accept E3 ;
      PUT_LINE ( " E3 accepted " ) ;
   end T3 ;
begin
   PUT_LINE ( " After begin in BUNCH" ) ;
   X := F1 ;
   T2.E2 ;
   X := F1 ;
   T3.E3 ;
   X := F1 ;
   T1.E1 ;
   PUT_LINE ( " At end of BUNCH" ) ;
exception
   when STORAGE_ERROR =>
      PUT_LINE ( "STORAGE_ERROR" ) ;
   when TASKING_ERROR =>
      PUT_LINE ( "TASKING_ERROR" ) ;
   when CONSTRAINT_ERROR =>
      PUT_LINE ( "CONSTRAINT_ERROR" ) ;
   when others =>
      PUT_LINE ( " Other exception" ) ;
end BUNCH ;
```

EXAMPLE. Program rewritten to add the STORAGE_SIZE rep specs

```
with TEXT_IO ; use TEXT_IO ;
procedure BUNCH is
   COUNTER : INTEGER := 0 ;
   X : INTEGER ;

   task type T1_TYPE is
      entry E1 ;
   end T1_TYPE ;
   for T1_TYPE'STORAGE_SIZE use 400*512;

   task type T2_TYPE is
      entry E2 ;
   end T2_TYPE ;
   for T2_TYPE'STORAGE_SIZE use 400*512;
```

```
task type T3_TYPE is
   entry E3 ;
end T3_TYPE ;
for T3_TYPE'STORAGE_SIZE use 400*512;

T1 : T1_TYPE;
T2 : T2_TYPE;
T3 : T3_TYPE;

function F2 ( PARAMETER : INTEGER ) return INTEGER is
   C : array ( 0 .. 40_000 ) of INTEGER ;
begin
   PUT_LINE ( " Inside function F2 " ) ;
   C ( PARAMETER ) := PARAMETER ;
   C ( PARAMETER + 1 ) := C ( PARAMETER ) + 1 ;
   return C ( PARAMETER + 1 ) ;
end F2 ;

function F1 return INTEGER is
   B : INTEGER := F2 ( COUNTER ) ;
begin
   PUT_LINE ( " Inside function F1 " ) ;
   COUNTER := COUNTER + 1 ;
   return COUNTER ;
end F1 ;


task body T1_TYPE is
   A : INTEGER := F1 ;
begin
   PUT_LINE ( " After begin in T1_TYPE" ) ;
   accept E1 ;
   PUT_LINE ( " E1 accepted " ) ;
end T1_TYPE ;

task body T2_TYPE is
   A : INTEGER := F1 ;
begin
   PUT_LINE ( " After begin in T2_TYPE" ) ;
   accept E2 ;
   PUT_LINE ( " E2 accepted " ) ;
end T2_TYPE ;

task body T3_TYPE is
   A : INTEGER := F1 ;
begin
   PUT_LINE ( " After begin in T3_TYPE" ) ;
   accept E3 ;
   PUT_LINE ( " E3 accepted " ) ;
end T3_TYPE ;
```

```
begin
   PUT_LINE ( " After begin in BUNCH" ) ;
   X := F1 ;
   T2.E2 ;
   X := F1 ;
   T3.E3 ;
   X := F1 ;
   T1.E1 ;
   PUT_LINE ( " At end of BUNCH" ) ;
exception

   when STORAGE_ERROR =>
      PUT_LINE ( "STORAGE_ERROR" ) ;
   when TASKING_ERROR =>
      PUT_LINE ( "TASKING_ERROR" ) ;
   when CONSTRAINT_ERROR =>
      PUT_LINE ( "CONSTRAINT_ERROR" ) ;
   when others =>
      PUT_LINE ( " Other exception" ) ;
end BUNCH ;
```

This much revision, simply to specify the size of the stack, seems unreasonable. Note that an alternative revision is to leave the task types and objects at their original positions but enclose them in a package, and to supply a use clause for the package. I haven't worked out the details of such a revision, but it was not the most obvious step to take, anyway.

To avoid such revision during porting, users with foresight ought to adopt the following "coding guideline":

> Task declarations must not be located in the later declarations. Single task declarations are not allowed.

It is clear to me that it is necessary for an implementation to accomodate application programs that have been written without such foresight. I feel certain that you will see implementations doing one of the following:

1. Giving users a way to specify the global default value to be used for the STORAGE_SIZE for tasks -- either via a pragma, or outside of the language (at link-time, say). For example, the following pragma might be permitted in the declarative part of the main program:

   ```
   pragma DEFAULT_TASK_STORAGE(...);
   ```

2. Inventing a pragma that serves as an alternative to the representation spec. The pragma would be allowed as a later declarative item, and would be allowed to apply to single tasks. For example:

pragma TASK_STORAGE(...);

I'd like to request guidance from the LMC in the best way to solve this problem. Is it possible that you could modify the language definitions so that the rep spec can be applied to single task declarations that are later declarations?

If you would leave it to implementations to solve this, I would favor pragma DEFAULT_TASK_STORAGE because it would allow users to force an implementation to adopt the default used by the prior implementation -- thus, affecting ALL task declarations that don't have the rep spec.

This seems the most useful thing to provide. Not only does it solve this particular problem, but it eliminates the need to give lots of rep specs for lots of other tasks. Would you agree, or have any comment?

Problem with naming of subunits 86-09-05 AI-00458/00 1

!standard 10.02 (05) 86-09-05 AI-00458/00
!class study 86-09-05 (provisional classification)
!status received 86-09-05
!topic Problem with naming of subunits

!summary 86-09-05

!question 86-09-05

!recommendation 86-09-05

!wording 86-09-05

!discussion 86-09-05

!appendix 86-09-05

****

!section 10.02 (05) D.Winter@cwi.nl 86-08-12 83-00792
!version 1983
!topic Problem with naming of subunits

Change last sentence of this paragraph to
The simple names of all subunits that have the same *parent unit* must be distinct identifiers.
Also paragraph 8 needs rewording in this style.
As it stands the combination of subunits A.B.D and A.C.D is disallowed, although their full names are distinct.

****

!section 10.02 (05) J. Goodenough 86-08-18 83-00793
!version 1983
!topic Naming of subunits
!reference 83-00792

The last sentence of this paragraph states the intent -- it is illegal to have subunits with full names A.B.D and A.C.D. Your suggestion was considered in the 1982 revision of the language (see comment #0889) and was not agreed to at the time. It is my recollection that the rule that exists in the Standard was proposed to make implementation support for subunits easier.

Your comment will be filed for review when changes to the language are being considered.

Allow non-integral powers for exponentiation 86-10-02 AI-00460/00 1

!standard 04.05.06 (00) (05,6) 86-10-02 AI-00460/00
!class study 86-10-02 (provisional classification)
!status received 86-10-02
!topic Allow non-integral powers for exponentiation

!summary 86-10-02

!question 86-10-02

!recommendation 86-10-02

!wording 86-10-02

!discussion 86-10-02

!appendix 86-10-02

****

!section 04.05.06 (05,6) R. Jones 86-09-08 83-00809
!version 1983
!topic Allow non-integral powers for exponentiation

I have a pocket calculator (Texas Instruments TI57 programmable bought in 1979 at UKL 24.00) that is quite capable of exponentiation by negative and fractional exponents and I feel that in consequence it is reasonable to expect any computer manufacturer to provide similar capability in their hardware.

Accordingly,l I suggest that the definition in paragraph 5 be amended to read as follows:

Operator Operation Left oprnd type Right oprnd type Result type

** expontiation any num type any num type some num type

It is possible that you may consider that the Right operand type should not include floating point types but this I leave to your discretion, though I would mention that my calculator has this facility.

It would also be necessary to reflect this alteration in paragraph 6.

Named associations for default array aggregates AI-00473/03 1
89-03-16 ST WI

| !standard 04.03.02 (06) 89-03-16 AI-00473/03
| !class study 89-02-27 (provisional classification)
!status work-item 87-04-16
!status received 86-10-13
!topic Named associations for default array aggregates

| !summary 89-03-16 (DRAFT)
|
| It was agreed by the ARG at its February 1989 meeting (9-0-1) that the topic
| of this commentary should be considered during the revision of Ada. In
| particular, the current rule specifying when named associations are allowed
| together with an OTHERS choice should be broadened to reflect the intent of
| the rule, namely, that such associations are allowed when no "sliding" of the
| index values is allowed. These are the contexts in which no implicit subtype
| conversion is applied to the aggregate. Such a rule would reflect the intent
| of the design.

!question 89-01-21 (DRAFT)

Consider the following example:

```
subtype S3 is STRING (1..3)
| procedure P (X : S3 := (1 => 'a', others => ' '))  -- legal? (should be)
    is ... end P;
...
P ( (1 => 'a', others => ' ') );                      -- legal
```

Although the aggregate used in the default expression for P is the same as the aggregate used in the call, the default expression appears to be forbidden by 4.3.2(6) even though the call is legal. 4.3.2(6) says:

> For an aggregate that [has an OTHERS choice and appears as the expression that follows an assignment compound delimiter or as an actual parameter or generic actual parameter when the corresponding formal parameter has a constrained array subtype], named associations are allowed for other associations only in the case of a (nongeneric) actual parameter or function result.

Since actual_parameter is a syntax term, the default expression in P's declaration cannot be considered an actual parameter in terms of 4.3.2(6)'s
| rule. On the other hand, 6.4.2(2) says that the value of a default expression is "used as an implicit actual parameter" in calls where the default is needed, so 6.4.2 suggests that default expressions are subject to the rules for actual parameters. If so, the default expression in the example is legal. Is the default expression legal or not?

A similar example can be written for a generic formal parameter:

```
generic
    X : S3 := (1 => 'a', others => ' ');          -- legal? (should be)
procedure Q;
...
procedure NQ is new Q ((1 => 'a', others => ' '));  -- legal (yes)
```

Is the default expression illegal even though the same aggregate is allowed as a generic actual parameter?

!response 89-03-16 (DRAFT)

The intention of the rules in 4.3.2(6) was to allow named associations with an OTHERS choice only when no subtype conversion is applied to the aggregate, i.e., to allow such a form only when no "sliding" of the bounds occurs. No sliding of the bounds occurs for an aggregate used as a default expression of a subprogram or generic formal parameter. Hence, an OTHERS choice could have been allowed together with named associations for aggregates appearing in such contexts. However, such usage is not allowed by 4.3.2(6). A revision of this rule should be considered for the next version of the language.

!appendix 89-01-24

****

!section 04.03.02 (06) J. Goodenough 86-10-05 83-00820
!version 1983
!topic Named associations for default array aggregates

Consider the following example:

```
procedure P (X : STRING (1..3) := (1 => 'a', others => ' ');
...
P ( (1 => 'a', others => ' ') );
```

Although the aggregate used in the default expression for P is the same as the aggregate used in the call, the default expression is illegal and the call is legal! 4.3.2(6) says:

> For an aggregate that appears in such a context [in particular, as an actual parameter or as the expression that follows an assignment compound delimiter] and contains an association with an OTHERS choice, named associations are allowed for other associations only in the case of a (nongeneric) actual parameter or function result.

Since actual_parameter is a syntax term, the default expression in P's declaration cannot be considered an actual parameter in terms of 4.3.2(6)'s rule, even though 6.4.2(2) says that a default expression is "used as an

implicit actual parameter" in calls where the default is needed. Since the contexts allowing the use of named associations together with an OTHERS choice does not include use as a default expression of a formal parameter, P's declaration is illegal. The call, however, is clearly allowed by 4.3.2(6).

The 4.3.2(6) rule would be equivalent to saying that named associations together with an OTHERS choice are allowed in contexts where "sliding" of bounds does not occur, if it were not for the fact that no sliding is allowed for aggregates used as default expressions of subprograms.

Was it intended for the aggregate in P's declaration to be considered legal?

****

!section 04.03.02 (06) G. Mendal 87-03-02 83-00908
!version 1983
!topic Error in comment 83-00820

!reference 83-00820, AI-00473

The example in AI-00473/00 contains a slight syntax error. Types of formal parameters of a subprogram must be type marks. Obviously, you only need to define such a subtype and use it as the formal type in the example.

****

!section 04.03.02 (06) Chuck Engle 89-01-24
!version 1983
!topic Results of checking compilers

The ARG requested that compilers be checked to determine their behavior on the cases of interest to this commentary. The following test cases were run with the indicated compilers. The results are indicated as follows:

A compiler accepted indicated statement or declaration
R compiler rejected indicated statement or declaration
* behavior was not the behavior expected by the recommendation
\- no results are available

Version numbers of the compilers are:

Dec Vax	1.5
Verdix	5.41
Alsys	3.2
Meridian	2.0
Telesoft	3.22A
Tartan	2.06

Rational	Delta 0
DDC	4.2

No tested compilers accepted example (1) since a similar case is explicitly checked by the ACVC (B43202B contains just such an example).

```
with TEXT_IO;
procedure TEST_473 is

  subtype S3 is STRING (1..3);
  procedure P (X : S3 := (1 => 'b', others => 'c')) is
                          -- (1) legal? (no)
  begin

    TEXT_IO.PUT_LINE("Test 473: Procedure P executed.  Results: ");
    TEXT_IO.PUT_LINE(X);
    TEXT_IO.NEW_LINE;
  end P;

begin  -- TEST_473
  TEXT_IO.PUT_LINE("End of Test 473.");
end TEST_473;
```

	(1)
DEC VaxAda (VMS)	R
Verdix (ULTRIX)	R
Alsys (MS-DOS)	R
Meridian (MS-DOS)	R
TeleSoft	R
Tartan	R
Rational	R
DDC	R

Case choises should not have to be static. 86-10-13 AI-00477/00 1

!standard 05.04 (05) 86-10-13 AI-00477/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic Case choises should not have to be static.

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 05.04 (05) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00835
!version 1983
!topic Case choises should not have to be static.

In a case statement choise values must be static, but not in a corresponding if statement. At least the requirement that a constant derived through a type conversion cannot serve in specifying range (doomed nonstatic by Ada) should be removed. Example:

```
package A is
   type A_T is range 1..10;
   A_C: constant A_T:= 2;
end A;

with A;
package B is
   type B_T is new A.A_T;
   B_C: constant B_T:= B_T(A.A_C);
end B;
package body B is
   V: B_T;
begin
   ...
   case V is
      when B_C => -- ERROR: expression not static!
      ...
```

```
    end case;
    if V = B_C then -- OK!
      ...
end B;
```

Why is the treatment different in the case and if statements? If the compiler can solve the if statement, why could it not solve the case statement, too?

Note that currently correct Ada programs compile and execute correctly after the proposed modification.

Referring to out-mode formal parameters to be 86-10-13 AI-00478/00 1

!standard 06.02 (05) 86-10-13 AI-00478/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic Referring to out-mode formal parameters to be allowed.

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 06.02 (05) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00832
!version 1983
!topic Referring to out-mode formal parameters to be allowed.

There is nothing more natural than to use, e.g. to check, the value being delivered out. For example:

```
procedure write(DATA: Item; STATUS: out Status_Type) is
begin
   MT.write(DATA, DATA'LENGTH, STATUS);
   if STATUS /= SUCCESS then
 -- ERROR: reference to out variable!
      log_error(STATUS);
 -- ERROR: reference to out variable!
   end if;
end write;
```

Experience has shown that the forbidden references to out-variables leads to bad programs. The following ways to solve the problem have been used:

- local parameters for intermediate processing and copying the actual parameter just before the return,

- and the use of inout-variables instead of out-variables.

The local variables' idea is not very good:

1. weakened performance due to additional copying,

2. programs become more complicated, which leads to more errors, since there are more (unnecessary) local variables, in particular because the same thing is represented by two objects (the local one and the one in the parameter list): the wrong one may easily be used, and the copying may even be forgotten in some situations.

The inout-variables' alternative is not much better. A tradition in programming languages has been that the mode of procedure parameters tells their role in the procedure: in-variables are used for input, inout-variables may have a value in the entering but it may be converted to another value in the exiting, and out-variables will only receive a value in the exiting. If one has to use inout-variables instead of out-variables (as they should be) the program logic is confused ("does the input value of this variable have some significance, or is it inout only due to the nasty feature of Ada?")

At Nokia Information Systems, the author has given instructions to his programmers to adopt the inout-variables alternative with trailing comments " -- out" at places where the false "inout" appears in the code. This choise seems less error prone and enables a natural way of processing data, until the problem is solved in Ada itself.

Note that currently correct Ada programs compile and execute correctly after the proposed modification.

Access type out-variables should be null befor 86-10-13 AI-00479/00 1

!standard 06.02 (06) 86-10-13 AI-00479/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic Access type out-variables should be null before call

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 06.02 (06) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00833
!version 1983
!topic Access type out-variables should be null before call

Allowing the reference to out-mode formal parameters (see previous proposition on 6.2(5)) requires that the parameter passing rules must be changed. Out-mode parameters shall not be copied in, but they shall have the value null in entering to the procedure. In this way illegal use of an out-mode access parameter can be prevented.

Note that currently correct Ada programs compile and execute correctly after the proposed modifications.

Visibility of predefined operators with derive 86-10-13 AI-00480/00 1

!standard 08.03 (14) 86-10-13 AI-00480/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic Visibility of predefined operators with derived types

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 08.03 (14) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00834
!version 1983
!topic Visibility of predefined operators with derived types

The visibility of operators +, -, =, /= vanishes when a type is derived from one in another package:

```
package A is
   type A_T is range 1..10;
end A;

with A;
package B is
   subtype B_T is A.A_T;
   V: B_T:= B_T'FIRST + 1; -- ERROR: + not visible!
end B;
```

There can be no doubt, what '+' means here, hence the program ought to be accepted as quite correct.

Note that currently correct Ada programs compile and execute correctly after the proposed modification.

Must standard input and output files be indepe 86-10-13 AI-00485/00 1

!standard 14.03 (05) 86-10-13 AI-00485/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic Must standard input and output files be independent?

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 14.03 (05) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00836
!version 1983
!topic Must standard input and output files be independent?

The Ada I/O validation suite determines that the standard input and output files must be independent of each other so that any operation of one of the files shall not affect the layout of the other, i.e. the colums, line, and page must remain unaltered. It would be more natural to have these files connected to the same layout description, so that the I/O system could visualize the layout as it is on a workstation window. It is suggested that the requirement of independency be dropped.

The TEXT_IO procedures end_of_page and end_of_ 86-10-13 AI-00487/00 1

!standard 14.03.04 (21) 86-10-13 AI-00487/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic The TEXT_IO procedures end_of_page and end_of_file

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 14.03.04 (21) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00837
!version 1983
!topic The TEXT_IO procedures end_of_page *and* end_of_file

The definition of the functions end_of_page (14.3.4(21)) and end_of_file (14.3.4(24)) is such that they return true even if there is still one (empty) line to be read on the page or in the file, respectively. This is not very logical.

Skipping of leading line terminators in get ro 86-10-13 AI-00488/00 1

!standard 14.03.06 (03) 86-10-13 AI-00488/00
!class study 86-10-13 (provisional classification)
!status received 86-10-13
!topic Skipping of leading line terminators in get routines

!summary 86-10-13

!question 86-10-13

!recommendation 86-10-13

!wording 86-10-13

!discussion 86-10-13

!appendix 86-10-13

****

!section 14.03.06 (03) H. Pohjanpalo/Nokia Inf. Systems 86-10-10 83-00838
!version 1983
!topic Skipping of leading line terminators in get routines

The skipping of leading line terminators in numeric (14.3.7(6,13) and 14.3.8(9,18)), character (14.3.6(3)), and string get (14.3.6(9)) routines in the TEXT_IO package does not seem well justified. In interactive I/O, these get routines cannot be used in ordinary applications. Default values are usually given as carriage returns:

```
...
put("Data length (default = 20) = ");
INTEGER_IO.get(LENGTH);
```

If the user strikes carriage return, the control remains in the get routine, the line terminators are only skipped. If the empty answer were defined to cause the exception DATA_ERROR, the problem could be solved in the application. With the current Ada the solution is adopting the get_line routine, but is there then very much use for the numeric, character, and string get routines? In practice almost all programmers in the beginning make erroneous programs of the type in the example, which shows that the operation is not what one would expect.

Use of national symbols and standards in an IS 87-01-13 AI-00510/00 1

!standard 02.01 (11) 87-01-13 AI-00510/00
!class study 87-01-13 (provisional classification)
!status received 87-01-13
!topic Use of national symbols and standards in an ISO standard

!summary 87-01-13

!question 87-01-13

!recommendation 87-01-13

!wording 87-01-13

!discussion 87-01-13

!appendix 87-01-13

****

!section 02.01 (11) Czechoslovakian ISO Member Body 86-11-15 83-00872
!version 1983
!topic Use of national symbols and standards in an ISO standard

In international standards, including this one, no references to national standards should be made and no national symbols be used. In this case, references to the ANSI standards and to ASCII and EBCDIC character codes are not permitted; references to ISO standards and to symbols used therein should be made instead.

National graphic character symbols, namely $ (dollar) should not be used. They are to be replaced by corresponding symbols adoped by ISO, namely, the currency symbol.

The definition of the language should not prevent using of national alphabets in certain parts of program:

- in identifiers
- in character literals
- in string literals
- in comments (although nothing is said in this Draft International Standard concerning the character set used in comments)

In these parts of program not only application of the ISO 646 but also application of other ISO standards (ISO 4873, ISO 2022, ISO 8859, ISO 6937) should be allowed.

We therefore propose that corresponding articles be modified:

- 2.1, 2.5, 2.6, 2.7, 2.10, 3.5.2, and Appendix C (type CHARACTER, package ASCII)

We agree with the opinion of the Secretariat of ISO/TC 97/SC 22 expressed in answering the Japanese comments in 1985-06-06 (J. L. Cote), that it will be sufficient to resolve these problems in the first revision of the resulting standard.

Fixed and Floating type Declarations needlessl 87-02-10 AI-00518/00 1

!standard 03.05.09 (02) 87-02-10 AI-00518/00
!class study 87-02-10 (provisional classification)
!status received 87-02-10
!topic Fixed and Floating type Declarations needlessly Different

!summary 87-02-10

!question 87-02-10

!recommendation 87-02-10

!wording 87-02-10

!discussion 87-02-10

!appendix 87-02-10

****

!section 03.05.09 (02) Terry Froggatt 86-12-11 83-00890
!version 1983
!topic Fixed and Floating type Declarations needlessly Different

If there is ever a major revision of Ada, the declaration of fixed and floating types should be unified: in both cases the programmer wants to give a fairly coarse indication of the minimum accuracy required. This is discussed in some more detail in my paper, "Fixed-Point Conversion, Multiplication, & Division, in Ada(R)", to appear shortly in Ada Letters.

Clearly, fixed-point types need a range, whereas this is optional for subtypes and floating-point types. The range should be used to determine the scale of the fixed-point type, such that (at least) one bound is (almost) represented by the endpoints of the underlying integer type.

The other thing that a fixed-point declaration has to do is to enable the implementation to decide which underlying integer type best meets the user's minimum accuracy requirements. This can be done by a "digits" clause just as for floating-point, and the number given should specify the number of digits required for the whle mantissa, not just the fractional part.

Thus, fixed and floating point type declarations for the same range and accuracy could be identical, even though fixed types give absolute accuracy whereas floating types give relative accuracy. Of course, some means of distinguishing a fixed type declaration from a floating one is needed. This could be done in true Ada style, by overloading an existing keyword:

fixed_accuracy_definition ::= ABS DIGITS static_simple_expression.

"Small" should be a power of two TIMES THE RAN 87-02-10 AI-00519/00 1

!standard 03.05.09 (05) 87-02-10 AI-00519/00
!class study 87-02-10 (provisional classification)
!status received 87-02-10
!topic "Small" should be a power of two TIMES THE RANGE

!summary 87-02-10

!question 87-02-10

!recommendation 87-02-10

!wording 87-02-10

!discussion 87-02-10

!appendix 87-02-10

****

!section 03.05.09 (05) Terry Froggatt 86-12-07 83-00886
!version 1983
!topic "Small" should be a power of two TIMES THE RANGE

Ada's default power of two scaling of "small" was a mistake. Power of two scaling is more of a distraction than an abstraction: it seems to be of very limited use. For serious embedded applications, range-related scalings are necessary: and in their absence programmers will sensibly use pure fractions.

With range-related scalings, we get maximum accuracy, we get range checks at minimum cost, and we avoid spurious scaling operations. Thus we get a cheap floating point that is both cheaper and better than with power of two smalls. And we get the scaled fractions of classical fixed-point working for use where this is appropriate whether or not we have floating hardware, such as angles with natural scalings and sensors with given scalings.

If there is ever a major revision of Ada, the right solution would be to make range-related scalings the default. There would then be no need for small representation clauses at all. (Anyone then wanting the current default scaling or a true delta-related scaling could achieve this by declaring a type with an appropriately expanded range followed by a subtype of it with the required range: no extra facility is needed).

Implementation of range-related scaling is in itself straighforward, but for Ada's counter-productive accuracy requirements.

These matters are discussed in some more detail in my paper, "Fixed-Point Conversion, Multiplication, & Division, in Ada(R)", to appear shortly in Ada Letters.

Fixed Point Subtypes inheriting Small 87-02-10 AI-00521/00 1

!standard 03.05.09 (14) 87-02-10 AI-00521/00
!class study 87-02-10 (provisional classification)
!status received 87-02-10
!topic Fixed Point Subtypes inheriting Small

!summary 87-02-10

!question 87-02-10

!recommendation 87-02-10

!wording 87-02-10

!discussion 87-02-10

!appendix 87-02-10

****

!section 03.05.09 (14) Terry Froggatt 86-12-05 83-00884
!version 1983
!topic Fixed Point Subtypes inheriting Small

3.5.9 (14) states, and 3.5.9 (16) clarifies, that a fixed point subtype S of a fixed point type T, inherits the Small of T if and only if that Small was specified by a length clause.

However, I imagine that most fixed point types will have a small length clause to ensure that Small is Delta rather than a power of 2. I cannot see small length clauses being used to produce a much smaller small than the delta, since this this can be achieved by using a smaller delta.

A subtype of T which inherits T's small can always be obtained by using a range constraint rather than a fixed-point constraint; so if a fixed-point constraint is given it should be honoured.

Thus, the Small of S should be the largest power of 2 times the Small of T that is not more than the Delta of the fixed-point constraint; regardless of whether T's Small was specified or defaulted.

Rounding up or down 87-03-11 AI-00526/00 1

!standard 04.06 (07) 87-03-11 AI-00526/00
!class study 87-03-11 (provisional classification)
!status received 87-03-11
!topic Rounding up or down

!summary 87-03-11

!question 87-03-11

!recommendation 87-03-11

!wording 87-03-11

!discussion 87-03-11

!appendix 87-03-11

****

!section 04.06 (07) Lee Phillips/Naval TSC 86-11-05 83-00909
!version 1983
!topic Rounding up or down

4.6(7) says "rounding may be either up or down" for conversion to an integer type when the operand is halfway between two integers. If rounding can be up or down, this can cause problems in porting programs from one compiler to another. This should be "up or down or controlable."

The portability of Ada programs may be a problem if this is not tightened up or controlable.

Resolving the meaning of an attribute name 87-03-11 AI-00529/00 1

!standard 09.09 (05) 87-03-11 AI-00529/00
!class study 87-03-11 (provisional classification)
!status received 87-03-11
!topic Resolving the meaning of an attribute name

!summary 87-03-11

!question 87-03-11

!recommendation 87-03-11

!wording 87-03-11

!discussion 87-03-11

!appendix 87-03-11

****

!section 09.09 (05) Dave Emery@MITRE-bedford.arpa 87-02-13 83-00899
!version 1983
!topic Resolving the meaning of an attribute name

Consider the following:

```
procedure foo;

task t is
  entry foo;
end t;

x : integer;
...
  x:= foo'count;      -- the point in question
```

Two compilers (Dec and Verdix) reject "foo'count" as being ambiguous, citing 4.1.4(3). Although this is probably a correct reading, it is not very 'user friendly', as it is clear that you cannot talk about the entry calls on a function. On the other hand, should this be OK?

```
task t is
  entry foo;
end t;

procedure bar renames t.foo;

x : integer
```

```
...
  x := bar'count;      -- what about this???
```

I would like to see the restrictions in 4.1.4(3) relaxed, so that obvious things work.

Declaring constant arrays with an anonymous ty 87-08-05 AI-00538/00 1

!standard 03.02 (09) 87-08-05 AI-00538/00
!class study 87-08-05 (provisional classification)
!status received 87-08-05
!topic Declaring constant arrays with an anonymous type

!summary 87-08-05

!question 87-08-05

!recommendation 87-08-05

!wording 87-08-05

!discussion 87-08-05

!appendix 87-08-05

****

!section 03.02 (09) J. Goodenough 87-07-07 83-00933
!version 1983
!topic Declaring constant arrays with an anonymous type

Consider the following example:

```
type ARR is array (INTEGER range <>) of INTEGER;
C1 : constant ARR := (1, 2, 3);                          -- legal
C2 : constant array (1..3) of INTEGER := (1,2,3);        -- legal
C3 : constant array (INTEGER range <>) of INTEGER :=
       (1, 2, 3);                                        -- illegal
```

Although the declarations of C1 and C3 both mention unconstrained array types, the declaration of C3 is illegal since the syntax does not allow an unconstrained array definition in this context. It seems non-uniform to allow C1 and C2 but not C3.

Need for static attributes of arrays and recor 87-08-05 AI-00539/00 1

!standard 04.09 (11) 87-08-05 AI-00539/00
!class study 87-08-05 (provisional classification)
!status received 87-08-05
!topic Need for static attributes of arrays and records

!summary 87-08-05

!question 87-08-05

!recommendation 87-08-05

!wording 87-08-05

!discussion 87-08-05

!appendix 87-08-05

****

!section 04.09 (11) Art Evans/Tartan Labs 87-06-26 83-00928
!version 1983
!topic Need for static attributes of arrays and records

Ada never permits an array type to be static. This fact follows from the third sentence of 4.9(11).

I see no reason why a type such as

```
type T is array(1..2) of integer;
```

should not be considered static. In particular, I find it reasonable that an array type be considered static if both of the following requirements are met:

> All indices must be static ranges; and
> the array element type must be static.

(A similar rule can be developed for record types.) If such a type is static, it should then follow that 'size of such a type (or an object of such a type) should be static.

This problem bit us when we tried to use the size of an array in a rep spec, where a static value is required.

The full declaration of a private type AI-00540/01 1
89-12-11 ST RE

| !standard 07.04.01 (01) 89-12-11 AI-00540/01
!class study 87-08-05
!status received 87-08-05
!topic The full declaration of a private type

!summary 87-08-05

!question 87-08-05

| !recommendation 87-08-05

!discussion 87-08-05

| !appendix 89-10-24

****

!section 07.04.01 (01) Art Evans/Tartan Labs 87-06-26 83-00929
!version 1983
!topic The full declaration of a private type

Ada does not permit renaming types, the syntax in *ARM 8.5* not including any way to do so. In general, the rationale for this lack is that a subtype declaration can be used instead. However, using a subtype does not work in all cases.

What I want is something like this:

```
with B;                        -- Package exporting a type.
package A is
   type T is private;          -- A private type.
   ...
private
   type T renames B.BT;        -- Illegal!
end A;
```

The renaming declaration is of course illegal. Moreover, the subtype mechanism doesn't help, since there's no such thing as a 'private subtype' declaration and a private type may not be completed with a subtype.

I see no clean way out of this problem. The problem can be solved (sort of) by completing the private type declaration with either of

```
type T is record
   F: B.BT;
end record;
```

or

```
type T is access B.BT;
```

I find both of these to be pretty ugly. Either affects all code that uses objects of the type.

Has anyone a better solution? This matter looks like a candidate for consideration when Ada is next revised.

****

!section 07.04.01 (01) Johan Backlund 87-06-27 83-00930
!version 1983
!topic The full declaration of a private type

!reference 83-00929

Have you tried this form (I guess you haven't...) ?

```
with Export_P;
package New_P is

   type New_T is private;
   ...
private

   type New_T is new Export_P.Old_T;

end New_P;
```

New_T becomes a so called "derived type" and LRM 3.4 tells you all about it.

****

!section 07.04.01 (01) Erhard Ploedereder 89-10-24 83-01309
!version 1983
!topic Completion of types

!summary
It would be convenient if incomplete and private types could be completed by a subtype declaration rather than a full (derived) type declaration.

!rationale
Completion by subtypes would avoid the necessity of frequent type conversions between derived types, where such type equivalence is appropriate.

File "append" capability proposed 87-08-05 AI-00544/00 1

!standard 14.02.01 (00) 87-08-05 AI-00544/00
!class study 87-08-05 (provisional classification)
!status received 87-08-05
!topic File "append" capability proposed

!summary 87-08-05

!question 87-08-05

!recommendation 87-08-05

!wording 87-08-05

!discussion 87-08-05

!appendix 87-08-05

****

!section 14.02.01 (00) David A. Smith 87-07-22 83-00938
!version 1983
!topic File "append" capability proposed

The following comment represents discussions that took place at the Nov 1986 and Jan 1987 meetings of the Ada Language Issues Working Group (ALIWG).

The question was raised at the Nov meeting as to whether the lack of an APPEND capability in TEXT_IO and SEQUENTIAL_IO was a serious problem for users of Ada. Those in attendance voted overwhelmingly that it was. Several approaches to solving the problem were presented, but the group split almost evenly between two, which were significantly different, as described below. Those present at the January meeting voted to recommend Solution One, below.

Solution One:

This approach involves adding a new procedure to TEXT_IO and SEQUENTIAL_IO of the form:

```
procedure APPEND (FILE : in out FILE_TYPE;
                  NAME : in     STRING;
                  FORM : in     STRING := "");
```

APPEND has the effect of opening FILE so that all previous contents are retained and subsequent calls to PUT or WRITE add their information at the end of the file. Note that there is no MODE parameter - APPEND assumes OUT_FILE.

There was some debate as to whether the APPEND should automatically create FILE if it does not already exist (a "friendly" APPEND), or whether it should fail ("unfriendly"). Sentiment was for the unfriendly version. Sentiment also favored a "clean" function for determining whether a named file already exists -- this is discussed in another comment.

The suggestion that a boolean parameter be added to indicate friendly or unfriendly opens was rejected. The possibility of adding yet another, friendly version of APPEND was left open.

Some participants voiced concern with the behavior of APPEND running on computers that support multiple versions of files within the host file system (such as VAX/VMS). The final consensus was that it did not really matter how the APPEND procedure was implemented, from a purely Ada point of view. It was pointed out, however, that a reasonable compiler vendor would provide a way for the programmer to achieve the desired results through the FORM parameter.

The biggest advantage to Solution One is that it is upward compatible in nature, that is, not a single line of Ada code already written would have to be modified. It adds the new capability without forcing massive changes to existing programs.

Solution Two

The second solution is to use the existing OPEN procedure but change the enumerated type FILE_MODE to include elements for all possible file interfaces. A sample of the idea is given below:

```
type FILE_MODE is (IN_FILE, OUT_FILE, CREATE_FILE, FRESH_FILE,
                   FRIENDLY_APPEND, UNFRIENDLY_APPEND);

   -- IN_FILE            open for reading, signal an error if the file
   --                    doesn't already exist
   --
   -- OUT_FILE           open for writing, signal an error if the file
   --                    doesn't already exist
   --
   -- CREATE_FILE        open for writing, signal an error if the file
   --                    already exists
   --
   -- FRESH_FILE         open for writing, delete the file if it
   --                    already exists (friendly CREATE)
   --
   -- FRIENDLY_APPEND    open for writing, retain old contents if the
   --                    file already exists, otherwise create a new
   --                    file
   --
   -- UNFRIENDLY_APPEND  open for writing, retain old contents if the
   --                    file already exists, otherwise signal an error
```

Notice that there is no need for the CREATE procedure under this scheme, since

creating a file is taken care of by one module explicitly and two others implicitly.

This scheme is very attractive because of the use of a single OPEN and is probably the way file interfacing should have been taken care of in the initial definition of the language. Its one major drawback is that it is a departure form the current file model and adopting it will cause much existing code to be changed.

Conclusion

The lack of an append facility is a significant drawback to Ada, felt by users who need a file-append capability and have no practical, portable way of getting it.

Procedure to find if a file exists 87-08-05 AI-00545/00 1

!standard 14.02.01 (00) 87-08-05 AI-00545/00
!class study 87-08-05 (provisional classification)
!status received 87-08-05
!topic Procedure to find if a file exists

!summary 87-08-05

!question 87-08-05

!recommendation 87-08-05

!wording 87-08-05

!discussion 87-08-05

!appendix 87-08-05

****

!section 14.02.01 (00) David A. Smith 87-07-22 83-00939
!version 1983
!topic Procedure to find if a file exists

The following comment represents discussions that took place at the Nov 1986 and Jan 1987 meetings of the Ada Language Issues Working Group (ALIWG).

It was agreed that a subprogram is needed in the three predefined I/O packages that, given a string, would indicate whether or not a file by that name exists. Its approximate form would be:

```
function FILE_EXISTS (FILENAME : in STRING) return BOOLEAN;
```

This upward-compatible function would provide a capability that is needed but currently lacking. The work-around now is to attempt to open the file, and then if that fails, immediately close it. If the file does not exist, an exception is raised:

```
with TEXT_IO;

function FILE_EXISTS (FILENAME : in STRING) return BOOLEAN is
  FILE : TEXT_IO.FILE_TYPE;
begin
  TEXT_IO.OPEN (FILE, TEXT_IO.IN_FILE, FILENAME);
  TEXT_IO.CLOSE (FILE);  -- only reached after successful open
  return TRUE;
exception
```

```
    when others => return FALSE;
  end FILE_EXISTS;
```

There are several flaws in this "home-grown" solution. First is that any program using FILE_EXISTS must "with" it separately from TEXT_IO. Furthermore, the above example only works with TEXT_IO files. Since SEQUENTIAL_IO and DIRECT_IO are generic, a generic FILE_EXISTS must be made available and instantiated in parallel. This is clumsy at best.

There was concern as to the exact semantics to be implemented. Semantics requiring an "open then close" implementation are not desired, since under UNIX and perhaps other systems, this would update the time stamp on the file.

An alternative semantics was suggested: "return true if an attempt to open would succeed". In this case, it was suggested that a more descriptive name would be appropriate. A true response to a "file exists" query may not be sufficient information, if the file cannot be opened for some other reason. The following name was suggested, although there was no agreement on the exact name:

```
function CAN_FILE_BE_OPENED (MODE : in FILE_MODE;
                             NAME : in string;
                             FORM : in string) return BOOLEAN;
```

To fully test if the file can be opened, this function requires the "form" and "mode" parameters.

Releasing heap storage associated with task type instances AI-00570/00 1
88-07-06 ST RE

!standard 04.08 (07) 88-07-06 AI-00570/00
!class study 88-07-06
!status received 88-07-06
!topic Releasing heap storage associated with task type instances

!summary 88-07-06

!question 88-07-06

!recommendation 88-07-06

!discussion 88-07-06

!appendix 87-11-12

****

!section 04.08 (07) R. W. Shore/Booz-Allen 87-11-12 83-00954
!version 1983
!topic Releasing heap storage associated with task type instances

The Ada compiler we are using does not allow heap space associated with an instance of a task type to be recovered under any circumstances. (Comment in a paper appearing in Ada Letters, Sept-Oct 1987, vii.5-106) [Such restrictions should be considered in the proposed revision of Ada. J. Goodenough]

Unique path name for subunits AI-00572/00 1
88-07-06 ST RE

!standard 10.02 (05) 88-07-06 AI-00572/00
!class study 88-07-06
!status received 88-07-06
!topic Unique path name for subunits

!summary 88-07-06

!question 88-07-06

!recommendation 88-07-06

!discussion 88-07-06

!appendix 88-03-15

****

!section 10.02 (05) D. Smith 88-03-15 83-00962
!version 1983
!topic Unique path name for subunits

The following comment represents discussion that took place at the January 1987 and December 1987 meetings of the Ada Language Issues Working Group (ALIWG).

Currently a subunit must have a unique simple name among all the subunits of a given ancestor library unit. It is proposed that this restriction be relaxed so that uniqueness is required only for the full expanded name of the subunit. It is proposed that the following (presently illegal) program be made legal:

```
package body P is
    procedure P1 is separate;
    procedure P2 is separate;
end P;

separate(P)
procedure P1 is
    procedure G is separate;
begin
    null;
end P1;

separate(P)
procedure P2 is
    procedure G is separate;
begin
    null;
```

```
end P2;
```

It is recognized that difficulties caused by the present rule can be worked around using renaming. The vote was unanimous in favor of this proposal.

Need a standard name for null address AI-00582/00 1
88-08-31 ST RE

!standard 13.07 (02) 88-08-31 AI-00582/00
!class study 88-08-31
!status received 88-08-31
!topic Need a standard name for null address

!summary 88-08-31

!question 88-08-31

!recommendation 88-08-31

!discussion 88-08-31

!appendix 88-07-07

****

!section 13.07 (02) David Emery 88-07-07 83-00984
!version 1983
!topic Need a standard name for null address

Most, if not all, compilers provide a named value that represents a null, invalid or zero address. This value is extensively used in interfacing Ada to other languages (especially C) or host operating systems. Since there is no standard (Ada predefined) name for this value, every vendor presents his own name. This causes portability problems for software that would otherwise be portable (i.e. compiles without any textual changes to the source code).

Recommend that the following be added to the predefined part of package SYSTEM:

```
NULL_ADDRESS : constant ADDRESS;
    -- value is implementation dependent
```

There is a (small, in my opinion) issue concerning machines that do not support the notion of a null address. In this case, I guess the thing to do is to permit them to omit this declaration. The intent is for machines that do support the concept, that there is a standard Ada name for the value.

Obviously the URG can address this, but I think this should be considered for inclusion in the Ada standard.

****

!section 13.07 (02) Robert I. Eachus 88-07-07 83-00985
!version 1983
!topic Need a standard name for null address

!references 83-00984 ST RE

I agree, but in this (1983) version of the standard, this is a URG issue, since such declarations are permitted but implementation defined. Also IMHO, this should be required of all implementations. It may require extra checking, but in general it is possible on all systems to return a value which is not a possible value for any legitimate use of ADDRESS clauses or 'ADDRESS. (The AdaEd interpreter of course had one value of type address, but that value would have been the appropriate value for NULL_ADDRESS.)

Robert I. Eachus

```
with STANDARD_DISCLAIMER;
use STANDARD_DISCLAIMER;
function MESSAGE (TEXT: in CLEVER_IDEAS) return BETTER_IDEAS is...
```

Restrict argument of RANGE attribute in Ada 9x AI-00584/00 1
88-09-02 ST RE

!standard 03.06.02 (02) 88-09-02 AI-00584/00
!class study 88-09-02
!status received 88-09-02
!topic Restrict argument of RANGE attribute in Ada 9x

!summary 88-09-02

!question 88-09-02

!recommendation 88-09-02

!discussion 88-09-02

!appendix 88-07-20

****

!section 03.06.02 (02) Keith Enevoldsen 88-07-20 83-00992
!version 1983
!topic Restrict argument of RANGE attribute in Ada 9x

If we take an Ada program written for one compiler and try to compile it on another compiler or analyze it with a general purpose source code static analysis tool (like a call tree or cross reference generator) we expect that it will either fail to compile (perhaps it uses LONG_INTEGER or an implementation-specific attribute) or it will successfully compile and all overloading will be resolved in the same way on both compilers.

However, we have discovered that in certain rare cases different implementations may legally resolve the same construct in different ways. As far as I can tell, these cases always involve the use of the RANGE attribute with an argument containing an attribute which has an implementation-defined value, like SIZE or DIGITS, or a named number defined in terms of one of these attributes.

In the following example, the call to procedure P will resolve to the first P if INTEGER is 16 bits, or to the second P if INTEGER is 32 bits.

```
with TEXT_IO; use TEXT_IO;
procedure TEST is
   type T is array(BOOLEAN, 11 .. 12) of CHARACTER;
   procedure P(A : BOOLEAN) is
   begin
      PUT_LINE("P - BOOLEAN");
```

```
    end;
    procedure P(A : INTEGER) is
    begin
      PUT_LINE("P - INTEGER");
    end;
  begin
    for X in T'RANGE(INTEGER'SIZE / 32 + 1) loop
      P(X);
    end loop;
  end;
```

We would like the RM to follow this general principle : An overloaded construct must not be successfully resolved in two different ways on two different implementations.

NOTE: We will have to exclude from this principal any constructs which involve implementation-specific attributes because two different implementations could conceivably choose the same name for two different implementation-specific attributes of different types.

One way to enforce this general principle is to add this special rule: The argument of the RANGE attribute must not contain any attributes with implementation-defined values, or any named numbers defined in terms of these attributes. The attributes which are allowed in universal integer static expressions, but which would be disallowed in the argument of the RANGE attribute are:

AFT, DIGITS, EMAX, FORE,
MACHINE_EMAX, MACHINE_EMIN, MACHINE_MANTISSA, MACHINE_RADIX,
MANTISSA, SAFE_EMAX, SIZE, WIDTH

This rule would be of the most benefit to developers of general purpose Ada tools (not compilers) which need to do overload resolution but may otherwise ignore implementation-dependencies.

Name of the "current exception" AI-00595/00 1
88-10-05 ST RE

!standard 11.04 (00) 88-10-05 AI-00595/00
!class study 88-10-05
!status received 88-10-05
!topic Name of the "current exception"

!summary 88-10-05

!question 88-10-05

!recommendation 88-10-05

!discussion 88-10-05

!appendix 87-08-19

****

!section 11.04 (00) David A. Smith 87-08-19 83-00942
!version 1983
!topic Name of the "current exception"

The following comment represents discussion that took place at the July 1986 meeting of the Ada Language Issues Working Group (ALIWG).

It has been noted by many users that it is frequently desirable in handling an exception to have access to a string representing the exception's name. This is useful for debugging purposes and for building diagnostics into a program. Work-arounds for this capability are awkward or impossible to build. While this feature is available in some implementations, the consensus was that it should be required of all implementations (like, for example, T'IMAGE for enumeration types). Many variations on this capability were discussed.

1) The string value could be accessed in a variety of ways:

a) Put(Exception'Image); -- attribute
-- This odd use of a reserved word seems unnecessary. There
-- doesn't seem to be anything appropriate for this string
-- to be an attribute of -- it would be meaningful as an attribute
-- of a task, but the main task in particular is not nameable.

b) Put(Standard.Exception_Name); -- function call
Put(System.Exception_Name); -- function call
Put(Current_Exception.Exception_Name); -- function call
Put(Exception_Name); -- function call
-- It was agreed this string should be returned by a zero-argument
-- function; the function could be its own library unit or could

-- be exported by an ad hoc package.
-- Packages Standard and System both seem inappropriate.
-- Agreement on subprogram and/or package names was lacking.

2) The string returned could be one of a number of things:

a) The simple name of the exception
b) The "fully qualified" name (this is not well defined)
c) The simple name plus the line number and compilation unit where the exception is defined

The sentiment favored more information than the simple name of the exception, but neither option b nor c is very satisfactory.

3) Two proposals were considered for the scope for invoking this function:

a) The function returns a non-null string when called from a point where the "RAISE;" statement is allowed (ie, without an explicit exception name). If called elsewhere, returns a null string.

b) The function can be called anywhere and returns a null string or "the most recent exception whose handling has not been completed", (assuming this is well defined).

It was suggested that syntactically restricting the point of call (option a) might be necessary in some implementations.

Option b was felt to be more "user friendly". A variation would be to return "the most recently raised exception, if any".

Why We Need Unsigned Integers in Ada AI-00600/00 1
88-11-08 ST RE

!standard 03.05.04 (00) 88-11-08 AI-00600/00
!class study 88-11-08
!status received 88-11-08
!topic Why We Need Unsigned Integers in Ada

!summary 88-11-08

!question 88-11-08

!recommendation 88-11-08

!discussion 88-11-08

!appendix 88-09-28

****

!section 03.05.04 (00) Ivar Walseth 88-08-19 83-01020
!version 1983
!topic Why We Need Unsigned Integers in Ada

<I received the following letter, and am taking the 'liberty' of forwarding it to Ada-Comment, since I think that the intent of the sender is to provide this kind of input.
Dave Emery>

Sivilingenior Kjell G. Knutsen, A.S.
P.O. Box 95
N-4520 SOR-AUDNEDAL
Norway

19th August 1988

Dave Emery
MITRE
MS A156
Bedford, MA 01730

Why We Need Unsigned Integers in Ada

During the last year I've managed a project where we are implementing communication protocols in Ada (protocols specified in the CCITT recommendations X.213, X.214, X.215, X.224, X.225, X.409, X.410, X.411, X.420).

In the communictions world they operate with octets. Each octet

contains 8 bits, and all bit combinations should be available. For this purpose it is of course possible to define the type:

```
type octet is range 0..255;
for octet'size use 8;        -- optionally
```

So far so good. The need for a standardized unsigned integer facility in Ada arises when we are using this octet type for generating checksums (according to X.224, Appendix I). This algorithm requires modulo 255 arithmetic. For this purpose we have of course implemented our own slow machine-independent arithmetic operators.

The next problem arises when the protocol specificiations says that bit number 6 has some special meaning (as defined in table 2 in X.409). To fetch (and store) this value we use logical operators such as "and" "or" and shift-functions. These are implemented in a compiler-dependent way. We could of course have used some tricky records or multiply and division operators, but we didn't find these solutions better when it comes to portability and performance.

In addition to the use of octets, other parts of the protocols need bitwise operations on bigger unsigned integers (ref. tag and length coding of X.409-units). My wish for Ada 9X is therefore either the 3 types unsigned_8, unsigned_16 and unsigned_32, where the number specifies the number of bits of the variable, or a more general type unsigned from which my own types might be created. In the latter case Ada should guarantee at least 32 bits are available in the type.

For the chosen type I'm hoping for the following functions and operators:

- arithmetic without overflow (appropriate modulo arithmetic)
- bitwise logical operators : and, or, xor, not
- shift-functions without bit rotation (optionally separate functions with the more rare shift-variants)
- operators for comparisons
- procedures for fetching and storing single bits
- conversion between unsigned and "normal" integer types

The various HW-manufacturers like to number the bits of a byte differently. In the CCITT world the octet has 8 bits numbered from 1 to 8 where the leftmost bit is MSB and has the number 8. I suggest the same numbering in Ada.

Best Regards,
per Siv.ing. Kjell G Knutsen A/S

/signed/
Ivar Walseth

****

!section 03.05.04 (07) 88-09-28 83-01021
!version 1983
!topic Unsigned Arithmetic

!reference AI-0597, 83-00974

Bryce lists the following goals for unsigned numbers in Ada: (my summary)

1. Non-negative integer range that exploits available hardware (and that supports full range address arithmetic)
2. Numeric literals in arbitrary bases, to the full range of the unsigned type.
3. Provide efficient support for modular arithmetic.
4. Provide straightforward and efficient logical operators (including shifts, rotates and masks) on bits of unsigned types.

I fully support the first three goals, but I cannot support the fourth. Unsigned integers are integers, and there is no arithmatic definition of shifts, rotates and masks on integers (signed or unsigned). On the other hand, I think it is probably necessary to support conversion between integers (unsigned and signed) and some representation of bit arrays on which shifts, rotates and masks can be supported. (this conversion should be more than Unchecked_Conversion, because there is no guarantee that U.C. does what you want.) I guess there needs to be a function that converts between integer types and appropriately sized bitmaps, something like:

```
type bitmap is array (positive range <>) of boolean;
pragma pack (bitmap);   -- NOTE:  this should work (i.e
                        --    bitmap(1..32)'size should be 32)

function to_bitmap (i : some_integer_type)
    return bitmap;      -- raises constraint_error if i'size >
                        -- bitmap'size.  does something TBD
                        -- (maybe zero fill) if i'size < bitmap'size

function to_integer (bitmap_32)
    return some_integer_type;   -- can raise constraint_error if
                                -- conversion would result in a
                                -- value out of the range of
                                -- some_integer
-- define appropriate shift, rotate and mask operations on
-- the bitmap type
```

There is no reason to restrict this conversion to unsigned integers.

I think I like Bryce's distinction between UNSIGNED_INTEGER and CARDINAL_INTEGER. My feeling is that these should be added to Standard, and that implementations should provide types like UNSIGNED_32 and CARDINAL_32, with efficient implementations of appropriate operations, in package SYSTEM. Such types are clearly system-dependent.

I guess the goals I'd set for unsigneds include the first 3 goals

provided by Bryce, plus a fourth goal:

4. (emery's goal) Types derived from unsigned types behave correctly and reasonably.

Here's a place where this would be important: Steve Litvintchouk has been looking at the Joint Frequency Hopping Specification. It has lots of bit-packed fields, like X is a 3 bit integer range 0..7. So I'd like to be able to do the following:

```
type X is new unsigned_integer range 0..7;
for X'size use 3;
```

and get all appropriate operations, including (in this case) modular arithmetic.

Why We Need Unsigned Integers in Ada
88-11-08

AI-00600/00 4
ST RE

Dave Emery
emery@mitre-bedford.arpa

Can't correctly read a file written with Text_IO AI-00605/00 1
88-11-22 ST RE

!standard 14.03.06 (00) 88-11-22 AI-00605/00
!class study 88-11-22
!status received 88-11-22
!topic Can't correctly read a file written with Text_IO

!summary 88-11-22

!question 88-11-22

!recommendation 88-11-22

!discussion 88-11-22

!appendix 88-11-16

****

!section 14.03.06 (00) David B. Kinder, BiiN 88-11-16 83-01034
!version 1983
!topic Can't correctly read a file written with Text_IO

If I write a multi-page file with Text_IO (e.g., after every 10 calls to PUT_LINE I call NEW_PAGE), then if I try to read the file with GET_LINE and test for END_OF_PAGE, the END_OF_PAGE never happens:

```
while not END_OF_FILE( FILE) loop
   if END_OF_PAGE( FILE) then
      PUT_LINE( "Now we're on page " & POSITIVE_COUNT'IMAGE( PAGE( FILE)));
   end if;

   GET_LINE( FILE, INPUT_LINE, LAST);
   -- do some processing
end loop;
```

The call to END_OF_PAGE will never return TRUE (except for an empty file). This is because the specification of GET_LINE says "Reading stops if the end of the line is met, in which case the procedure SKIP_LINE is then called." [LRM 14.3.6/13] Unfortunately, SKIP_LINE also skips pages if the line terminator is immediately followed by a page terminator [LRM 14.3.4/8], which is just the case we've got after every tenth line.

This is the opposite behavior of the other GET routines (such as for characters or for integers) where leading page terminators are skipped but not any following page terminators.

-- David Kinder, BiiN

Floating point machine attributes inadequate AI-00609/00 1
88-12-13 ST RE

!standard 13.07.03 (05) 88-12-13 AI-00609/00
!class study 88-12-13
!status received 88-12-13
!topic Floating point machine attributes inadequate

!summary 88-12-13

!question 88-12-13

!recommendation 88-12-13

!discussion 88-12-13

!appendix 88-12-08

****

!section 13.07.03 (05) J. Goodenough 88-12-08 83-01052
!version 1983
!topic Floating point machine attributes inadequate

The current set of machine attributes for floating point types is inadequate to describe all properties of the machine numbers. In particular, the attributes do not say whether a denormalized representation is possible. For example, suppose a machine allows denormalized representations, e.g., suppose the smallest representable value is:

2#0.001#E-128

In canonical, i.e., normalized, notation, this value is

2#0.100#E-130

Should MACHINE_EMIN therefore be -130? In this case, this value does not imply that a full set of canonical numbers is representable for such an exponent value, e.g.,

2#0.101#E-130

is not a representable value. If MACHINE_EMIN is understood to mean the smallest exponent value for which full MACHINE_MANTISSA precision is available, then -130 is not the appropriate value for MACHINE_EMIN; 2#0.101#E-130 is not, in fact, representable. On the other hand, if -128 is used as the value of MACHINE_EMIN, then 2#0.100#E-128 is not, in fact, the smallest representable positive number.

This shows that MACHINE_EMIN and MACHINE_MANTISSA together are not adequate to characterize the representation of machine values. At least one additional attribute is needed, and the definition of MACHINE_EMIN needs to be clarified.

Can't declare a constant of a 'null' record type. AI-00681/00 1
88-12-21 ST RE

!standard 04.03.01 (01) 88-12-21 AI-00681/00
!class study 88-12-21
!status received 88-12-21
!topic Can't declare a constant of a 'null' record type.

!summary 88-12-21

!question 88-12-21

!recommendation 88-12-21

!discussion 88-12-21

!appendix 88-12-14

****

!section 04.03.01 (01) D. Emery (emery@mitre.org) 88-12-14 83-01254
!version 1983
!topic Can't declare a constant of a 'null' record type.

Consider the following "abstraction":

```
package ABSTRACTION is
  type T is private;
  Null_T : constant T;
private
  type T is record
    null;
  end record;
  Null_T : constant T := ???????
end ABSTRACTION;
```

There is no way to initialize the object Null_T, because there is no way to generate an expression that is of type T. Actually, there is one way, but this way is very ugly:

```
Bogus_Object : T;
Null_T : constant T := Bogus_Object;
```

One can only hope that the compiler doesn't generate any storage for records of this type. Furthermore, it is clear that in some sense the vlaue of Null_T is very undefined, which goes against the whole idea of declaring a constant. (This is true of any type where a constant is declared using an uninitialized type. The difference here is that this is the only choice I have to initialize Null_T.)

A potential solution is

```
Null_T : constant T := (others => 3);  -- pick any number....
```

The problem with is that the 'others' clause can't be used here because 4.3.1(1) requires that 'others' represent at least one component.

In part it seems to me that this is due to the dual use of "null". In the declaration, it represents 'nothing'. But in an expression, it represents 'null access value'. I think in some respects

```
Null_T : constant T := T'(null);
```

would make sense, if it weren't for the fact that 'null' in this case must clearly represent a 'null access value', and not 'nothing'.

A third reasonable alternative would be:

```
Null_T : constant T := T'();
```

However, that is pretty ugly, too, as well as potentially harmful to parsers.

Overall, this is not a major problem, but is a significant surprise. This should be considered for Ada 9X, but not as a top priority item.

dave emery
emery@mitre.org

Attibutes SAFE_LARGE and SAFE_SMALL should be static. AI-00812/00 1
89-03-07 ST RE

!standard 04.09 (08) 89-03-07 AI-00812/00
!class study 89-03-07
!status received 89-03-07
!topic Attibutes SAFE_LARGE and SAFE_SMALL should be static.

!summary 89-03-07

!question 89-03-07

!recommendation 89-03-07

!discussion 89-03-07

!appendix 88-10-27

****

!section 04.09 (08) P. Miller (SofTech) 88-10-27 83-01255
!version 1983
!topic Attibutes SAFE_LARGE and SAFE_SMALL should be static.

The attributes SAFE_LARGE and SAFE_SMALL are defined by the base type. The base type must always be static. These attributes should be considered static even when they are applied to a nonstatic subtype.

Communication which is not allowed AI-00832/00 1
89-08-22 ST RE

!standard 13.09 (01) 89-08-22 AI-00832/00
!class study 89-08-22
!status received 89-08-22
!topic Communication which is not allowed

!summary 89-08-22

!question 89-08-22

!recommendation 89-08-22

!discussion 89-08-22

!appendix 89-08-14

****

!section 13.09 (01) B. A. Wichmann and J. Dawes 89-08-14 83-01300
!version 1983
!topic Communication which is not allowed

If an Ada program calls a subprogram written in another language by means of pragma INTERFACE, is the program erroneous if communication is achieved other than via parameters and function results? It would seem reasonable to communicate via a file, perhaps in additionto parameters and function results.

Access 'out' parameter as attribute prefix AI-00840/00 1
89-11-19 ST RE

!standard 04.01 (04) 89-11-19 AI-00840/00
!class study 89-11-19
!status received 89-11-19
!topic Access 'out' parameter as attribute prefix

!summary 89-11-19

!question 89-11-19

!recommendation 89-11-19

!discussion 89-11-19

!appendix 89-24-08

****

!section 04.01 (04) Hans Hurvig 89-24-08 83-01328
!version 1983
!topic Access 'out' parameter as attribute prefix

!reference 6.2(5), A62006D.ADA, B62006C.ADA

The rule in 4.1(4) forbidding an access 'out' parameter as a prefix for any attribute is overly restrictive.

Its justification is that such an appearance can imply that the access value is dereferenced in connection with appropriateness:

'CALLABLE, 'FIRST, 'LAST, 'LENGTH, 'RANGE, and 'TERMINATED.

But using an access 'out' parameter as a prefix for other attributes is harmless, and making it illegal is a distinct loss of functionality.

For instance, 'ADDRESS is quite well-behaved for any 'out' parameter, and it is very odd indeed to single out those that happen to have an access type.

I'd argue that the rule in 4.1(4) should simply be deleted as redundant; paragraph 6.2(5) says that an 'out' parameter cannot be read, and using an access object as a prefix for an appropriate-attribute constitutes reading, because it refers to the denoted entity, thereby making it illegal.

Legality of Programs with Impl.-Defined Pragmas AI-00850/00 1
89-12-11 ST RE

!standard 02.08 (08) 89-12-11 AI-00850/00
!class study 89-12-11
!status received 89-12-11
!topic Legality of Programs with Impl.-Defined Pragmas

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 02.08 (08) Erhard Ploedereder 89-10-24 83-01314
!version 1983
!topic Legality of Programs with Impl.-Defined Pragmas

!summary
02.08 (8) should be altered to allow for implementation-defined pragmas that may render text outside such pragmas illegal. However, no Ada implementation may require the presence of such pragmas in Ada programs.

!rationale
There are numerous examples of implementation-defined pragmas whose purpose is a user-provided guarantee to adhere to certain restrictions. E.g., tasking related pragmas that promise certain characteristics of an entry.

It is counter-productive to require that the compilation must succeed even if such assertions are violated. The user should be warned as early as possible (by non-acceptance of his/her program) that the given assertions have been violated.

"=" as a basic operation AI-00851/00 1
89-12-11 ST RE

!standard 03.03.03 (02) 89-12-11 AI-00851/00
!class study 89-12-11
!status received 89-12-11
!topic "=" as a basic operation

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 03.03.03 (02) Erhard Ploedereder 89-10-24 83-01318
!version 1983
!topic "=" as a basic operation

!summary
It should be seriously considered whether equality ("=", "/=") should be reclassified to be a basic operation, with a special rule that, for limited types, the definition of "=" provides the definition of this basic operation.

!rationale
Very frequently, the sole cause for a 'use'-clause in Ada programs is to obtain direct visibility to the equality operation, which for all but limited types cannot be hidden by a user-provided function definition. These 'use'-clauses are detrimental to code readability and lead to potential overload resolution problems (ambigutity rules).

The write-around of renaming equality locally is ugly and often not applied.

Yet, the rules of the language make it completely obvious that, for non-limited types, equality can bind only to the predefined operation, so that direct visibility of the type declaration and its implicitly declared equality cannot possibly alter the meaning of the operation. (ARM 6.7 (4+5)). For limited types, one could either stay with the current rule of requiring direct visibility, or one could limit the opportunity to declare equality to the same declarative part in which the declaration of the limited type occurs.

Exiting blocks AI-00852/00 1
89-12-11 ST RE

!standard 05.07 (00) 89-12-11 AI-00852/00
!class study 89-12-11
!status received 89-12-11
!topic Exiting blocks

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 05.07 (00) Erhard Ploedereder 89-10-24 83-01308
!version 1983
!topic Exiting blocks

!summary
It should be possible to exit (named) blocks.

!rationale
Extending the functionality of the exit statement in this way would be benefitial to code legibility.

INLINE should not apply to all overloads AI-00853/00 1
89-12-11 ST RE

!standard 06.03.02 (03) 89-12-11 AI-00853/00
!class study 89-12-11
!status received 89-12-11
!topic INLINE should not apply to all overloads

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 06.03.02 (03) Erhard Ploedereder 89-10-24 83-01317
!version 1983
!topic INLINE should not apply to all overloads

!summary
6.3.2 (3), second sentence, should be rescinded. Pragma INLINE should apply only to the closest preceding subprogram of the specified name.

!rationale
The current rule that pragma INLINE applies to all overloaded subprograms of the given name in the same declarative part is inconvenient and creates maintenance problems.

It is inconvenient, because

```
procedure foo(X: my_special_type); --- with a body of 1000 lines

procedure foo(X: integer); -- with a 1-line body
pragma INLINE(foo); --- OOPS ! gets the first foo, too
```

(To resort to renaming to restrict the applicability of the pragma is a horrible solution.)

It is problematic, because

the original code...

```
package bar is
    --- 500 lines of specifications....
    procedure foo(X: integer); -- with a 1-line body
```

```
        pragma INLINE(foo); --- (1)
    end bar;
```

may have to be modified to become

```
    package bar is
        procedure foo(X: my_special_type); --- with a body of 1000 lines
        --- 500 lines of specifications....
        procedure foo(X: integer); -- with a 1-line body
        pragma INLINE(foo);
    end bar;
```

Unless one scans the entire specification for INLINE pragmas, there is the danger of unintentional inlining of newly introduced subprograms.

Private Types are too private AI-00854/00 1
89-12-11 ST RE

!standard 07.04 (00) 89-12-11 AI-00854/00
!class study 89-12-11
!status received 89-12-11
!topic Private Types are too private

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 07.04 (00) Erhard Ploedereder 89-10-24 83-01319
!version 1983
!topic Private Types are too private

!summary
There should be a way for "friendly packages" to gain access to the information contained in the private part of a package.

!rationale
It is a recurring problem in the formulation of interface specifications that some types are pervasive throughout the interfaces offered. These types may have to be (limited) private to the user of the interfaces. Yet, it can be exceedingly inconvenient to combine all interfaces whose implementation requires knowledge of the internals of these types in a single package to establish the necessary visibility into the internals of the private type. Rather, a grouping by functional area or capabilities is desirable. However, such capability packages cannot be implemented cleanly, due to the obvious visibility problems. Furthermore, many of the employed implementation schemes falter when these private types are needed as actual parameters to generic units required for the implementation of the types themselves and visible to the users of the interfaces.

Examples of large interfaces that had to contend with these problems are MIL-STD-1838(a), PCTE Ada Binding, X-Window binding and others.

Current work-arounds range from unchecked_conversion techniques to hidden type implementation packages, assumed to be known only to the implementors of the interfaces but not the user of the interfaces. The former is obviously fraught with danger; the latter quickly becomes unwieldy and

requires numerous type conversions. Neither addresses the mentioned issue of generic instantiations with such private types cleanly.

A possible and rather simple solution would be to extend context clauses of packages such that "friendly access" to the private part of an imported unit can be established, e.g.,
with foo; -- normal non-private visibility
with private bar; -- grants visibility to private part of bar as well
package friend is ....

Secondary units as implicit specifications AI-00855/00 1
89-12-11 ST RE

!standard 10.01 (06) 89-12-11 AI-00855/00
!class study 89-12-11
!status received 89-12-11
!topic Secondary units as implicit specifications

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 10.01 (06) Erhard Ploedereder 89-10-24 83-01316
!version 1983
!topic Secondary units as implicit specifications

!summary
The notion of 10.01 (6) that a secondary unit without previously specification acts both as a library unit and secondary unit should be altered to make these implicit specifications "first-class citizens", so that compilation of such a subprogram declaration creates both a library and a secondary unit.

!rationale
The current semantics of 10.01 (6) introduces a "wart" in the library concept. In connection with obsolescence rules, it leads to unnecessary recompilations. Consider the compilation sequence

```
procedure foo is begin ...-- 1. body
                end foo;
with foo;
package Q is ... end Q;

procedure foo is begin ...-- 2. body
                end foo;
```

Here, the rules of 10.3 (seem to) require that Q is now obsolete, although there is obviously no good reason for it. (There are ACVC checks of this ilk, despite the last sentence of 10.3(5).)

If one always required separately compiled specifications or, better,

specified the semantics of compiling secondary units without previous specifications as implicitly generating such specifications (rather than serving both as library and secondary units), these problems and user inconveniences would disappear.

Obsolete optional bodies AI-00856/00 1
89-12-11 ST RE

!standard 10.03 (05) 89-12-11 AI-00856/00
!class study 89-12-11
!status received 89-12-11
!topic Obsolete optional bodies

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 10.03 (05) Erhard Ploedereder 89-10-24 83-01313
!version 1983
!topic Obsolete optional bodies

!summary
The current interpretation of 10.03(5), requiring successful linkage of an Ada main program in the presence of obsolete optional bodies should be reversed. It should, at least, be made implementation-dependent whether or not an Ada system rejects such linkage (or execution).

!rationale
If optional bodies become obsolete then there is a distinct danger that linkage of the program will yield an incorrectly executing Ada program. Ada implementors should have the option of diagnosing this situation with an error message, aborting the link (or execution) attempt, and requiring that the obsolete body be explicitly deleted from the library or else recompiled.

priorities of interrupts AI-00857/00 1
89-12-11 ST RE

!standard 13.05.01 (02) 89-12-11 AI-00857/00
!class study 89-12-11
!status received 89-12-11
!topic priorities of interrupts

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 13.05.01 (02) Erhard Ploedereder 89-10-24 83-01311
!version 1983
!topic priorities of interrupts

!summary The determination of the priority of the interrupt (rendezvous) vis-a-vis software rendezvous should be left application-dependent.

!rationale

The rule in 13.05.01(2) that interrupts act as entry calls issued by a hardware task whose priority is higher than the priority of any user-defined task is ill-conceived and may have dire consequences in applications.

There is no reason to believe that, in an embedded system, the handling of an arbitrary hardware interrupt should take precedence over currently executing software tasks.

Functions implemented in Machine Code AI-00858/00 1
89-12-11 ST RE

!standard 13.08 (00) 89-12-11 AI-00858/00
!class study 89-12-11
!status received 89-12-11
!topic Functions implemented in Machine Code

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section 13.08 (00) Erhard Ploedereder 89-10-24 83-01312
!version 1983
!topic Functions implemented in Machine Code

!summary
It would be convenient for the user, if code statements were also allowed in function bodies. It should then be considered to permit return statements in such bodies along with code statements.

!rationale
Casting code inserts that are conceptually functions into a procedure mold is inconvenient and often less efficient.

By providing return statements in a sequence of code inserts, the user could be relieved of the necessity to know the respective calling conventions used by the specific compiler.

Pragma LIST AI-00859/00 1
89-12-11 ST RE

!standard B (06) 89-12-11 AI-00859/00
!class study 89-12-11
!status received 89-12-11
!topic Pragma LIST

!summary 89-12-11

!question 89-12-11

!recommendation 89-12-11

!discussion 89-12-11

!appendix 89-10-24

****

!section B (06) Erhard Ploedereder 89-10-24 83-01310
!version 1983
!topic Pragma LIST

!summary
Pragma LIST should be downgraded to an optional element of the language.

!rationale
Pragma LIST is an anachronism is this day and age of on-line, screen-oriented program maintenance. Furthermore it contradicts the explicit exclusion of the form or content of listings in the standard (1.1.1(k)).

INDICES

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

KEY TECHNICAL TERMS

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY NUMBER

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY TITLE

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY SUBMITTER

REVISION REQUEST BY ORGANIZATION

REVISION REQUEST BY ORGANIZATION

Bofors Electronics AB
3-65, 3-79, 3-85, 3-92, 3-182, 5-29, 6-106, 6-111, 7-30, 9-75, 9-76, 9-77, 9-113, 9-131, 9-132, 9-152, 11-29, 11-30, 13-32

Booz-Allen
17-116

Brandeis University
3-33, 10-16, 15-10

C.S. Draper Laboratory
9-45, 17-73

California Space Institute
9-82

Canadian AWG #010
13-38

Canadian AWG #011
13-39

Canadian AWG #014
9-89

Canadian AWG #013
8-40

Canadian AWG #012
11-36

Centrum voor Wiskunde en Informatica
3-88, 3-192, 4-33, 4-40, 4-99, 4-126, 6-44, 7-67, 9-68, 9-156,

Chrysler Technologies Airborne Systems
3-241, 8-57, 9-104, 9-133, 10-34, 10-76, 13-43,

Clemson University
2-2, 2-24, 3-37, 3-205, 4-7, 4-8, 4-23, 4-42, 5-4, 6-7, 6-18, 6-22, 6-48, 7-4, 7-23, 10-14, 11-2, 11-16, 12-2, 12-55, 14-6, 14-9, 15-21, 16-42, 16-43, 16-46, 16-92

Contraves AG
6-102, 7-17

CONVEX Computer Corporation
11-46

Czechoslovakian ISO Member Body
17-97

REVISION REQUEST BY ORGANIZATION

DALIN SOFTWARE
17-61, 17-63

DARPA/ISTO
8-45, 16-112

Delco Systems Operation
6-13

Direction de l'Electronique et de l'informatique
9-85

DY-4 Systems Inc.
16-65, 16-66, 16-67

ENEA Data AB
2-8, 4-80, 4-130

Ericsson Radar Electronics AB
9-143, 11-22, 11-45, 3-222, 16-57, 16-59, 16-61

ESI
6-39, 6-104, 9-150, 11-37, 13-42, 13-45, 13-60, 14-28, 16-71

Fairleigh Dickinson University
9-51, 9-52

Ferranti Computer Systems Limited
6-2, 13-2

Florida International University
3-189, 4-101, 5-44, 8-36, 10-40, 11-40, 14-46, 14-48, 14-51, 15-16, 16-63, 16-91

Florida State University
3-2, 3-35, 9-4, 9-9, 9-54, 13-20, 13-26

GA Technologies
17-31

GE Aerospace GCSD
3-20, 4-6, 6-12

General Dynamics
1-16, 1-22, 3-38, 3-71, 3-75, 3-77, 3-78, 3-80, 3-83, 3-116, 3-119, 3-151, 3-155, 3-157, 3-159, 3-176, 3-184, 3-202, 3-235, 3-242, 3-252, 3-260, 3-267, 3-268, 7-15, 7-39, 7-42, 7-43, 8-10, 4-45, 4-46, 4-62, 4-63, 6-24, 6-26, 6-38, 8-61, 8-62, 9-58, 9-103, 9-122, 9-125, 10-28, 10-36, 10-37, 10-62, 11-18, 13-29, 13-40, 13-46, 13-48, 13-56, 13-71, 14-12, 14-31, 14-32, 14-53, 15-15, 15-25, 16-20, 16-21

REVISION REQUEST BY ORGANIZATION

REVISION REQUEST BY ORGANIZATION

Intermetrics, Inc.
3-53, 3-55, 3-56, 3-57, 3-59, 4-22, 6-8, 6-20, 6-32, 6-34, 6-41, 7-9, 7-12, 8-12, 8-41, 9-35, 9-70, 10-10, 10-18, 11-13, 11-23, 12-3, 13-8, 17-7

IPSYS plc
3-113, 6-95, 7-40, 8-15, 10-60, 12-17, 12-19

ITT Avionics
3-29, 3-101, 4-9, 9-129, 13-10, 13-86

Japanese Member Body/ISO
17-53

Jet Propulsion Laboratory
3-111, 8-28, 14-35, 14-37

Link Flight Simulation Division of 15-CAE-Link Corporation
1-18, 2-29, 3-118, 3-163, 3-211, 4-103, 5-43, 6-60, 9-108, 9-110, 9-111, 10-52, 11-17, 14-10, 15-19, 16-52, 16-53, 16-56,

LMSC 5T 40
4-4, 4-11

Lockheed Missiles and Space Company
3-22

Logicon, Inc.
1-11, 1-12, 1-13, 1-17, 2-7, 2-31, 3-73, 3-129, 3-203, 4-26, 4-29, 4-55, 5-25, 5-27, 5-28, 5-30, 6-28, 7-72, 8-46, 16-22, 16-23, 16-26

Magnavox Electronic Systems Company
3-24, 6-10, 9-21, 9-41, 9-42

Martin Marietta Astronautics Group
4-14, 7-63, 9-7, 10-46, 11-15, 12-23

MBB
9-96

MBB BmbH
16-16

McDonnell Douglas Electronic Systems Company
3-86, 3-209, 3-259, 3-265, 4-86, 6-112, 6-113, 7-33, 8-32, 9-157, 12-32, 16-106, 16-108, 16-103, 16-104

McKee Consulting
8-14

REVISION REQUEST BY ORGANIZATION

MITRE
9-60, 17-105

National Physical 10-Laboratory
1-5, 2-26, 3-160, 10-39, 16-34

Naval Weapons Center
2-23, 3-117, 3-194, 7-49, 8-47, 9-127, 9-141, 10-5, 12-52, 13-18, 13-50, 15-13, 17-9

Naval Sea Systems Command
9-105, 9-120, 9-134, 9-146

Naval Security Group Detachment
1-9, 6-115, 13-55

Naval TSC
17-104

Naval Air Development Center
12-50

NDRE
7-3

Nimble Computer Corporation
3-237, 3-167, 3-169, 4-35, 4-68, 4-71, 4-73, 4-75, 4-113, 5-10, 6-36, 13-8, 16-39

Nokia Information Systems
17-88, 17-90, 17-91, 17-92, 17-93, 17-94, 17-95, 17-96

Peregrine Systems, Inc.
8-58

Planning Research Corporation
9-78, 13-36

Plessey Research
1-15, 3-162, 4-104, 4-105, 6-23, 6-50, 7-14, 8-50, 8-55, 9-56, 9-57, 9-145, 10-56, 10-70, 13-44, 13-57, 15-22, 15-28, 16-48, 16-49, 16-50, 16-51

PRIOR Data Sciences Ltd.
2-11, 3-103, 3-121, 3-124, 3-128, 9-155, 13-81, 15-32, 16-27

R & D Associates
1-7, 3-97, 3-183, 4-52

R.R. Software, Inc.
4-47, 5-8, 6-81, 9-4, 10-74, 11-8, 12-5

REVISION REQUEST BY ORGANIZATION

Rational
1-24, 2-32, 3-63, 3-81, 3-91, 3-100, 3-107, 3-153, 3-174, 3-185, 3-191, 3-195, 3-196, 3-224, 3-226, 3-257, 3-266, 4-37, 4-39, 4-43, 4-54, 4-57, 4-78, 4-83, 5-13, 5-20, 5-32, 5-51, 5-52, 6-58, 6-85, 6-100, 6-101, 7-68, 7-74, 8-30, 8-38, 8-53, 9-118, 9-139, 10-42, 10-72, 11-25, 11-26, 11-31, 11-38, 12-34, 12-43, 12-48, 12-57, 12-63, 12-65, 14-41, 16-94, 16-95, 16-96, 16-97, 16-101, 16-124,

Raytheon Company
9-48

Rekencentrum der Rijksuniversitreit
2-4

Rockwell/CDC
7-26, 8-21, 8-19, 9-72

Royal Military College of Canada
13-69

RSRE
1-19, 6-30, 10-25

S.A. OFFIS N.V.
3-89, 3-212, 3-216, 3-228, 3-245, 4-59, 4-64, 4-66, 4-82, 6-53, 6-89, 6-92, 7-34, 7-53, 7-64, 9-114, 9-158, 10-48, 10-66, 12-39, 12-41, 12-58, 12-62, 14-16, 14-20, 14-43, 14-44

SAIC
2-22, 3-49, 3-230, 9-19

School of Information Science
3-7

Scientific Computing Associates, Inc.
16-117

SD-Scicon PLC
3-10, 3-14, 3-45, 4-19, 4-84, 4-87, 4-92, 6-16, 7-7, 8-4, 10-12

Sema Group plc
7-5, 10-8, 10-22, 16-11

Shell Development Company
11-51, 11-52, 16-114, 16-115

Siemens AG ZFE F2 SOF3
2-3, 3-5, 3-19, 6-4, 14-7, 14-8

SINGER
17-13, 17-16

REVISION REQUEST BY ORGANIZATION

REVISION REQUEST BY ORGANIZATION

CHANGE PAGES
FOR
ADA 9X PROJECT
REVISION REQUEST REPORT
AUGUST 1989

The changes noted below are being made at the request of the original submittors. Change pages are attached

Remove	Insert
pp. 3-24/3-25	pp. 3-24/3-25 Change 1
pp. 3-26/3-27	pp. 3-26 Change 1/3-27
pp. 11-2/11-3	pp. 11-2/11-3 Change 1
pp. 11-4/11-5	pp. 11-4 Change 1/11-5

VISIBILITY OF BASIC OPERATIONS ON A TYPE

DATE: December 5, 1988

NAME: David Brookman

ADDRESS: Magnovox Electronic Systems Company
1313 Production Road
Department 542
Fort Wayne, IN 46808

TELEPHONE: (219) 429-4440
E-mail: CONTR22ONOSC-TECR.Arpa

ANSI/MIL-STD-1815A REFERENCE: 3.3.3, 8.1(10), 8.3(7)

PROBLEM:

Basic operations on a type, defined in a package specification, are not directly visible in another module which imports that package. If the basic operations are defined as infix operations, such as "=", then the infix notation cannot be used in the other module.

IMPORTANCE: ADMINISTRATIVE

These forces the programmer to use one of the three workarounds listed below.

CURRENT WORKAROUNDS:

1. Place a "Use" clause in the module importing the package. This ruins the traceability of identifiers. Therefore this is a very poor workaround.

2. Use the fully qualified form of the operations as a prefix operation. This reduces the readability of the resulting code.

3. Rename the operation so that it becomes directly visible. This is the best workaround, but it still is not ideal. It forces ugly rename clauses and may cause errors. If "<" is renamed to ">", the error could be quite difficult to detect.

POSSIBLE SOLUTIONS:

1. Make the basic operations directly visible.

2. Provide a "half-use" clause that only makes basic operations directly visible.

MULTIPLE TYPE DERIVATIONS

DATE: March 17, 1989

NAME: Mats Weber

ADDRESS: Swiss Federal Institute of Technology
EPFL DI LITh
1015 Lausanne
Switzerland

TELEPHONE: 0041 21 693 11
E-mail: madmats@elma.epfl.ch

ANSI/MIL-STD-1815A REFERENCE: 3.4

PROBLEM:

The type derivation mechanism has undesirable effects when multiple types declared in the same package visible part are derived. For example:

```
package Graph_Handler is

        type Vertex is private;
        type Arc is private;
        function New_Arc (Initial, Final : Vertex) return Arc;
        function Initial (Of_Arc : Arc) return Vertex;
        function Final (Of_Arc : Arc) return Vertex;
private

        ...

end Graph_Handler;

...

type Node is new Graph_Handler.Vertex;
        --      derives subprograms
        --      function New_Arc (Initial, Final : Node) return Graph_Handler.Arc;
        --      function Initial (Of_Arc : Graph_Handler.Arc) return Node;
        --      function Final (Of_Arc : Graph_Handler.Arc) return Node;

type Edge is new Graph_Handler.Arc;
        --      derives subprograms
        --      function New_Arc (Initial, Final : Graph_Handler.Vertex) return Edge;
        --      function Initial (Of_Arc : Edge) return Graph_Handler.Vertex;
        --      function Final (Of_Arc : Edge) return Graph_Handler.Vertex;
        --      What we need is
        --      function New_Arc (Initial, Final : Node) return Edge;
        --      function Initial (Of_Arc : Edge) return Node;
        --      function Final (Of_Arc : Edge) return Node;
```

IMPORTANCE: IMPORTANT

CURRENT WORKAROUNDS:

Use a lot of type conversions.

POSSIBLE SOLUTIONS:

Add multiple type derivations to the language, with a syntax like type (Node, Edge) is new (Graph_Handler.Vertex, Graph_Handler.Arc); which would derive the desired subprograms.

DERIVED TYPES

DATE: April 21, 1989

NAME: Stewart French

ADDRESS: Texas Instruments
P.O. Box 869305 MS 8435
Plano, TX 75086

TELEPHONE: (214) 575-3522

ANSI/MIL-STD-1815A REFERENCE: 3.4(17)

PROBLEM:

Current use of derived types is clumsy; current syntax requires renaming of derived types and operations.

IMPORTANCE: IMPORTANT

CONSEQUENCES:

Implementors will avoid use of derived types due to verbose declaration requirements of derived types.

CURRENT WORKAROUNDS:

Avoidance of construct.

POSSIBLE SOLUTIONS:

PASS EXCEPTIONS AS PARAMETERS OR ACCESS EXCEPTION'S 'IMAGE

DATE: January 15, 1989

NAME: William Thomas Wolfe

ADDRESS: Home: 102 Edgewood Avenue #2
Clemson, SC 29631 USA

Office: Department of Computer Science
Clemson University
Clemson, SC 29634 USA

TELEPHONE: Home: (803) 654-7030
Office: (803) 656-2847
E-mail: wtwolfe@hubcap.clemson.edu

ANSI/MIL-STD-1815A REFERENCE: 11.

PROBLEM:

Cannot pass exceptions as parameters, or access an exception's 'IMAGE.

CONSEQUENCES:

It is not possible to program a procedure ASSERT which takes as parameters a Boolean expression and an exception, and raises the desired exception if the Boolean expression turns out to be false. It is also not possible to generate warning messages to the effect that a particular exception was handled by the program if the name of the exception is not known in advance (e.g., in an others clause).

CURRENT WORKAROUNDS:

The ASSERT procedure can raise Assertion_Failed, and the call to the ASSERT procedure can be encapsulated in a local block which raises the desired exception upon encountering Assertion_Failed. However, this is unnecessarily verbose and tends to defeat the purpose of having an ASSERT procedure, since half of the logic involved in processing the assertion must appear outside the procedure call.

There is no workaround to the problem of finding out the image of an exception, in the general case. One could sit there and type when Some_Very_Long_And_Descriptive_Exception

=> PUT ("Some_Very_Long_And_Descriptive_Exception was handled..."); for every assertion whose name could be determined in advance, with obvious penalties in terms of code readability and cost.

POSSIBLE SOLUTIONS:

Revise ANSI/MIL-STD-1815A such that exceptions can be passed as parameters and such that exceptions have a 'IMAGE attribute.

FINDING THE NAME OF THE CURRENTLY RAISED EXCEPTION

DATE: April 7, 1989

NAME: Tony Orme

ADDRESS: ATM Computer
AEG (UK) Ltd.
Engineering Division
Eskdale Road
Winnersh
Wokingham
Berkshire
RG11 5PF
UK

TELEPHONE:

Country Code	44
Direct line	734 441419
Via Switchboard	734 698330 ex124
FAX	734 441397

ANSI/MIL-STD-1815A REFERENCE: 11.

PROBLEM:

In several common situations, it is important to be able to know the name of the currently raised exception, even when outside its scope.

These common situations are:

1. In on-line (not real-time) systems, where a human operator may have a choice of actions (say Retry, Abort or Detour) based on the exception raised;

2. Test harnesses and similar software should be able to print exceptions raised, perhaps with time, task id and other pertinent data, in reports;

3. Anywhere that anonymous exceptions may be handled. For example, in using a commercial package where some exceptions are not exported in the top level package spec (it may be objected that a commercial company should not supply such packages, but it is unwise overly to rely on a "shouldn't"). For tracing and other purposes, it is necessary to know what the exception was and where it was raised.

The problem is actually a composite of two slightly different problems:

a. In 1 and 2 above, the range of exceptions may be known, but an easy way displaying the current one is required (i.e., a case statement is cumbersome and awkward to maintain in this context);

b. In 3 above, an unknown exception may be raised, over which there is no control. Three has to be a way of discovering its identity.

IMPORTANCE: IMPORTANT

Effect on Current Applications: Possible solutions (see below) will have no impact on current applications.

Consequences of Non-implementation: If this amendment is not actioned, the penetration of Ada outside the military/aerospace sector may be impeded.

CURRENT WORKAROUNDS:

The D.G./Rolm ADE Ada compiler provides a non-standard package called CURRENT_EXCEPTION, with a single function NAME returning a string. (If there is no currently raised exception, a null string is returned.) However, in D.G./Rolm sites, this feature is frequently not used because it is completely non-portable.

POSSIBLE SOLUTIONS:

Because the raised exception may be out of scope, a means of returning a string rather than an exception is required.

The question of redeclared exceptions should also be considered. While it may be possible, except in unnamed blocks, to provide the name of the place where the exception was declared, it is felt on balance to be unnecessary.

Four possible solutions are:

1. A function to return the required information might be provided as a standard library unit. This has the merit of being relatively straightforward to implement.

2. A task attribute might be defined such as T'EXCEPTION. An extension would have to be made to permit enquiry on the main program. While this form allows wider possible use than solution 1, and is eminently suitable for situation 2 above, it is probably unnecessary complex.

3. The existing attribute IMAGE might be coupled with a predefined enumeration type, e.g., EXCEPTION_RAISED'IMAGE. This is really very similar to solution 1.

4. A string could be declared in the form of an (optional) parameter of an exception handler, e.g.,

```
exception (S : string)
  when
```

   There would be no corresponding actual parameter as exception handlers are not explicitly called. The situation where no exception has been raised does not arise. This may be close to an elegant solution.

IMPLEMENTATION OF EXCEPTIONS AS TYPES

DATE: May 30, 1989

NAME: David Papay

ADDRESS: GTE Government Systems Corp
PO Box 7188 M/S 5G09
Mountain View, CA 94039

TELEPHONE: (415) 694-1522
E-mail: papayd@gtewd.af.mil

ANSI/MIL-STD-1815A REFERENCE: 11., 3., 6., 7., 8., 12.

PROBLEM:

Currently, exceptions are distinct entities. There is no way to "group" related exceptions and handle them collectively, nor is there a way to declare a general "class" of exceptions and declare individual exceptions of that class. Finally, there is no way to pass exceptions are parameters to generic units, or to subprograms.

IMPORTANCE: IMPORTANT.

CURRENT WORKAROUNDS:

The current language rules allow multiple exceptions to be handled by one handler using exception choices, so this does solve part of the problem.

POSSIBLE SOLUTIONS:

Implement exceptions as types (in specific, as limited private types). Certain parts of the Ada syntax would change as follows:

3.3.1(2) type_definition ::=
...
| exception_type_definition

11.1 (2) (?) exception_type_definition ::= **exception**;

11.2 (2) exception_choice ::=
...
| **in** subtype_indication

where the base type of the subtype indication is an exception type.

Thus, the current syntax for declaring exception would still be legal:

QUEUE_OVERFLOW : **exception**;