LEVEL II

ON THE SUITABILITY OF ADA FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

July 1980

By: Richard L. Schwartz
P. M. Melliar-Smith
Project Scientists

Computer Science Laboratory

Sponsored by:

Defense Advanced Research Projects Agency (DoD
ARPA Order No. 2894

Monitored by:

U.S. Army Research Office

Under Contract No. DAAG29-79-C-0216
(SRI Project 1019)

Effective 28 Sept. 1979 thru 27 Sept. 1982

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI I
TWX: 910-37

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (19) 17127.1-A-EL (18) ARO
2. GOVT ACCESSION NO.: AD-A090790
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): (6) On the Suitability of ADA for Artificial Intelligence Applications
5. TYPE OF REPORT & PERIOD COVERED: (9) Technical rept.
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): (10) Richard L. Schwartz, P. M. Melliar-Smith
8. CONTRACT OR GRANT NUMBER(s): (15) DAAG29-79-C-0216, ARPA Order-2894
9. PERFORMING ORGANIZATION NAME AND ADDRESS: SRI International, Menlo Park, CA 94025
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (12) 36
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709
12. REPORT DATE: (11) Jul 80
13. NUMBER OF PAGES: 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

artificial intelligence
programming languages
system development

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report summarizes the results of our analysis of the suitability of the Department of Defense (DOD) programming language Ada for Artificial Intelligence (AI) applications. As Ada is expected to become a major programming language in the 1980's with widespread usage in both the military and commercial sectors, a reasonable question to be asked is whether Ada could benefit the efforts of AI system development or transfer. A large store of language and environment support

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT CONTINUED

tools are expected to appear within the next several years. The ability for the AI community to avail themselves of such technology would serve the needs of both AI systems developers and the ultimate recipients of the developed technology. In this report, we consider the suitability of the Preliminary Ada Language Specification, ~~as described in [1]~~, either as a language for the development of AI systems or as a language for the transfer of already-developed AI technology. We focus our attention on the language capabilities required to support AI paradigms and their realization in Ada.

SRI International

ON THE SUITABILITY OF ADA FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

July 1980

By: Richard L. Schwartz, Computer Scientist
P. M. Melliar-Smith, Senior Computer Scientist

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

U.S. Army Research Office
Research Triangle Park, NC 27709

Attention: Richard O. Ulsh

Under Contract No. DAAG29-79-C-0216
(SRI Project 1019)

Approved:

Jack Goldberg, Director
Computer Science Laboratory

David H. Brandin, Executive Director
Computer Science and Technology Division

SRI International

Table of Contents

ACKNOWLEDGMENTS

This evaluation has benefited greatly from the advice and insight of our consultants Peter Hibbard, Carl Hewitt, Jerry Feldman, Bob Balzer, Danny Bobrow, and Rod Burstall. Our communication with Jean Ichbiah and Paul Hilfinger has also contributed to our understanding of Ada and of the attendant language design decisions. Input to the project was also provided by members of the Artificial Intelligence Center at SRI, primarily Harry Barrow, Lynn Quam, and Marty Tenenbaum.

1. Scope of Report

This report summarizes the results of our analysis of the suitability of the Department of Defense (DOD) programming language Ada for Artificial Intelligence (AI) applications. As Ada is expected to become a major programming language in the 1980's with widespread usage in both the military and commercial sectors, a reasonable question to be asked is whether Ada could benefit the efforts of AI system development or transfer. A large store of language and environment support tools are expected to appear within the next several years. The ability for the AI community to avail themselves of such technology would serve the needs of both AI systems developers and the ultimate recipients of the developed technology.

In this report, we consider the suitability of the Preliminary Ada Language Specification, as described in [1], either as a language for the development of AI systems or as a language for the transfer of already-developed AI technology. We focus our attention on the language capabilities required to support AI paradigms and their realization in Ada.

2. Executive Summary

Our conclusion is that Ada, as it now stands or with any small set of changes, would not be suitable as a mainstream artificial intelligence research language. But, with some relatively modest extensions to Ada within the spirit of the language, it would be possible to translate a substantial proportion of AI algorithms from the research language in which the algorithm was originally developed into Ada. Such translation would not be a literal transcription but would be rather closer to a reimplementation of the program, retaining the complex heuristic algorithms that provide the artificial intelligence.

The report does not regard the unsuitability of Ada as an AI research language as being in any way a condemnation of Ada. Ada was designed for programming production-level real-time embedded computer applications, and the report is strongly supportive of the design of Ada for that purpose. There are some fundamental, and necessary, mismatches between the philosophy of design behind Ada as a standard DOD embedded computer programming language and that appropriate for a research language to support AI paradigms. The report recognizes that the addition of yet more features to Ada could detract from the suitability of the language for embedded systems development.

For some AI applications, the report considers that it may be possible to develop and refine the algorithm in Lisp, Sail, or some other AI language, and then translate, or perhaps more accurately, reimplement, the final version of the program into Ada. It is felt that Ada would make a strong contribution in providing a standard medium of expression for the remainder of the life cycle of the AI program. The completed Lisp or Sail program would act as a prototype for the coding of the "production" Ada program. Such a recoding would be accomplished by a software house familiar with AI paradigms and both the AI language and Ada. One must realize of course that, perhaps to an even greater extent than with embedded systems programs, there may never be a "completed" version of an AI program. Thus there remains the problem of updating the Ada version of the AI program to reflect the continuing evolution of the AI development version.

Clearly, in theoretical terms, almost any programming language is expressive enough to support the recoding of a Lisp or Sail program. In practical terms, however, the target language must be capable of expressing the AI algorithm in a manner that produces a readable, reasonably efficient final program, since it is the final program that must be maintained over the life-cycle. As maintenance has been shown to account for about 60% of the life-cycle cost of a program, this is a crucial requirement. The report suggests several extensions to Ada that would facilitate the expression of a wider range of AI algorithms.

The report presents the following crucial deficiencies of Ada, critical to the clear expression of AI paradigms:

- ☐ Procedures as Values,
- ☐ Garbage Collection,
- ☐ Additional Tasking Control, and
- ☐ Consistent Definition of Parameter Binding.

Without these changes to Ada, some programs will not be expressible within Ada and others will lead to obscure coding techniques in order to obtain the desired effect.

The report presents the following desirable but non-essential capabilities, that would contribute to the clearer expression of higher-level concepts:

- ☐ True Abstract Data Type Facility,
- ☐ Expanded Exception Handling, and
- ☐ Partial Parameterization.

Of the above identified issues, the report recommends that serious consideration be given to the following extensions to Ada, while remaining within its original objectives and spirit:

- ☐ procedures as parameters,
- ☐ a pragma requiring the presence of garbage collection,
- ☐ greater generality in type initialization,
- ☐ generic types,
- ☐ parameters to the instantiation of a type.

3. Characteristics of AI Programs and Programming Languages

As with many fields of research, it is difficult, if not impossible, to characterize the large and amorphous body of research calling itself artificial intelligence research. It is not the intent of the current study to address this question. Rather, we focus on the narrower technical question of the characteristics and language requirements of a sampling of systems being developed for image understanding, natural language understanding, and distributed computation. Many of our conclusions stemming from analysis of these areas of research will carry over to other AI research directions as well. Having acknowledged the difficulties of characterizing the entirety of artificial intelligence, we proceed with our analysis based on our chosen areas of AI research.

3.1. Application Characteristics

If a single characteristic of AI research and systems development could be isolated, it would probably be that of an *algorithm in flux*. AI programs, in general, have a far longer development and redesign cycle than that for the intended Ada community, and a far shorter period of use and maintenance. Frequently, this is coupled with considerable initial uncertainty as to the objectives and methods to be used, and thus a greater degree of evolution over the life-cycle of the program. It is common for there to be personal involvement and continuing support by the individual developer throughout the life of the program, and large programming teams are rare. This tends to lead to program practices emphasizing writeability over readability and thus tends to discourage program transfer.

Typically, an AI project is attempting to solve a problem involving great numbers of alternatives and considering large amounts of input data. In such areas, it is not possible to investigate every possible alternative in order to determine the "best fit". Rather, it is necessary to use *heuristic techniques* to make best guesses as to the proper decision to take at any given point. Classically (within the appropriate time frame), expansion of promising alternatives has taken place in a serial manner, using *breadth first* or *depth first* expansion strategies. If it becomes apparent that a previous decision was incorrect, the program *backtracks* to the decision point and expands a different decision path. More recently, parallel processing techniques have been used to achieve multi-programming or true multi-process solutions to the exploration of multiple alternatives.

It is characteristic for these heuristic techniques to be developed and honed on the fly. As flaws in each technique are discovered by exercising the system, changes are made and the perturbations to the system are observed. Based on these observations, further refinement of the algorithm is made. Frequently, alternative versions of an algorithm are constructed and dynamically bound in the system in order to probe the characteristics of each.

During AI systems development, data structures evolve as changes in the algorithm are made and as more is learned about the nature of the problem and about the developing approach. Associative and attribute access to information stored in data structures are frequently used structuring. These allow functional access to data without requiring knowledge of the exact data structure. Later, as the system characteristics become better understood, it may become desirable to specify more exactly the data structuring to be used.

Another structuring tool frequently employed is the embedding of algorithms within data structures. By allowing procedures to be used as values storable within data structures, the accessing algorithm can be closely coupled with the relevant data. As procedures are treated as storable values of a procedural data type, it is also common to pass procedures as parameters to manipulate and control data access. As a consequence, a general framework may be provided for parameterized transformation of a given general structure (e.g., for an arbitrary function to be performed on a standard tree or graph structure).

4. Programming Language Characteristics

In our discussion, we concentrate on two of the principal languages in use today for programming AI applications: Lisp [21] and Sail [29] . Each language enjoys widespread use and supports an active user community.

Lisp, developed in the early 1960s by McCarthy, is a language uniformly based on the manipulation of list structures. Programs and data are both represented as list structures. Consequently, it is possible to create functional abstractions on the fly, embed them in other data structures, and pass them as arguments to other abstractions. Functional abstraction is expressed in the language by lambda calculus [20] list expressions.

A large percentage of AI systems research carried out in Lisp centers around the Interlisp programming system [18], a *complete programming environment* built around Lisp. The environment includes a structure-oriented editor, a "programmer's assistant", a "do-what-I-mean" (not "what I say") package, a complete backtrackable transaction history, the ability to interpret, incrementally compile or block- compile, extensive debugging facilities, and various file and program management capabilities.

Within the Interlisp environment, there is no real distinction between user program and environment -- each environment function is a user-callable routine. It is thus possible for the application program to call the interpreter, the program editor, or the run-time debugger. This allows, for example, the user program to interrogate its own run-time context -- examining its run-time stack, nesting height, etc. As a consequence, programs developed in such an environment may become inseparable from their environment.

In addition to Interlisp, several other sophisticated programming environments have been built around Lisp, notably MACLISP [23] and CADR LISP [6].

Sail, another major AI programming language, is a reasonably high-level programming language based on Algol 60, and designed specifically for the DEC-System-10 computer. Unlike Interlisp, it is oriented toward a more conventional compiler-based system. It does, however, provide a set of execution-time debugging routines. In addition to the usual Algol features, Sail includes complete access to the DEC-10 I/O facilities, a flexible macro facility, event-based task management primitives, a user modifiable error handling facility, an associative data base, backtracking, and interrupt facilities. The language includes the standard assortment of built-in data types and a limited notion of user-defined types. One of the most important and widely used aspects of Sail is its very efficient STRING management facility, which includes a fast garbage collection algorithm. Sail is a moderately strongly typed language, providing some compile-time type checking but still allowing type breaching to occur.

Within Sail there is a distinction between program and data. It therefore is not possible to create and evaluate algorithms on the fly (as one can with EVAL in Lisp). Because Sail allows procedure variables, however, it is possible to embed procedures in data structures, pass them as parameters, etc.

Using the LEAP associative data base feature of the language, it is possible to store and retrieve information by attribute, without having to specify explicitly a data structure representation. As more is

learned about the problem domain, one can refine this into a more efficient representation (assuming that the machine hardware does not support associative memory access).

Sail, in general, provides a reasonably high-level language framework while allowing the user to delve into implementation details where necessary. It is possible to modify the system-provided memory allocation routine, interrogate the internal representation of data, and perform user-defined type conversion. Many of these capabilities are possible only because Sail is defined specifically for implementation with the DEC-10 architecture. The Sail language specification indicates the bit-level representation of each of the higher-level program features, thus allowing the user lower-level access to objects, while obviously eliminating any hope of a machine independent program. Herein lies a major difference between Sail and Ada. Despite the fact that both language are, in the general sense, derivatives of the Algol 60 tradition and share common control and data structuring tools, there is a fundamental difference in the level of implementation control that the languages allow. This distinction is explored in Section 5.

Mainsail [24], a MAchine INdependent version of Sail, was developed to satisfy the needs of those wishing to transfer the style of Sail programming to other machine architectures. It includes many of the characteristics of Sail but lacks the ability to delve into implementation details to the same extent as is possible in Sail. Several fundamental deficiencies of Mainsail, such as the lack of procedures as values, have limited its use thus far. There are hopes however that a redesigned superset version of Mailsail will soon appear.

5. Unsuitability of Ada as a General AI Research Language

Fundamentally, the unsuitability of Ada as a general AI research language can be traced to its determination to enforce a particular programming discipline. There is a goal of the designers of Ada (mandated in fact by the Ironman [19] and Steelman [31] Requirements) to foment an environment "encouraging good programming practices". The programming style imposed by Ada is not imposed arbitrarily, but is the result of years of research into programming methodology by the computer science community. While not everybody is agreed, and of course many might differ about the actual details of the language, the great majority of the computer science community strongly support the ideas on which Ada is based. All the comparisons known to us, using skilled programmers and non-trivial but well-understood programs, between highly disciplined programming languages such as Ada, Euclid, Mesa, or Simula, and less disciplined languages such as C, BCPL, or Fortran, have shown a clear advantage to the more highly disciplined languages.

Such attempts to mandate a programming style are *proper and beneficial* for the targeted Ada user community, systems programmers in software teams developing and maintaining computer software. In such environments, Ada is a major improvement for both the systems programmer and the program manager. Systems programmers who previously developed systems in Jovial, CMS-2, or other military-provided languages, will generally find Ada to be a major improvement. Ada provides control structures and data structuring tools not found in their previous languages and, because of the high degree of redundant specification in the language, it is able to provide the programmer with more

consistency checks than before. For the program manager, Ada contributes toward making programming an engineering discipline rather than an art by providing more standard ways of expressing solutions to design problems. The additional consistency checks which can be performed by the programming environment also provide more assurance that the coding of an algorithm is correct. Thus, Ada should provide significant gains for both the systems programmer and program manager.

The above characteristics of Ada, beneficial for a production environment, make it unsuitable as a general AI research language. As discussed in Section 3, AI research usually involves working within a problem domain for which techniques and approaches are developed on the fly. In such cases, there isn't always a clear understanding in advance of all that will be necessary for problem solution. AI systems being developed are experimental systems, and as such, additional freedom must be given to the developer of the system. Since it is difficult to provide just exactly the right degree of freedom, current AI languages cannot constrain the programmer's means of expression, but rather give him considerable expressive power to allow him the most natural means of expression. It is this role that AI languages like Lisp and Sail play.

Because of the intense redevelopment and change cycle that an AI program experiences, it is crucial that the language allow, and the programming environment support, incremental compilation. This is a language issue to the extent that the language should not preclude effective incremental compilation by excessive coupling of program segments. It appears that Ada language definition may preclude the development of an effective incremental compilation strategy. This is due in part to the partial ordering on the compilation sequence of modules imposed by the separate compilation facility. Such an imposed ordering is an environmental concern rather than a language concern and does not belong in the language specification. Whether such partial orderings on the compilation sequence are in fact necessary to support program modification is a function of how the environment is implemented.

For the reasons given above, we believe that Ada would be inadequate as a general AI research language.[1] It deliberately does not permit the flexibility necessary to support the type of development process inherent in the bulk of artificial intelligence research. The possible role of Ada is then as a target language for the expression of completed AI systems. An analysis of Ada with this role in mind is given in the following three sections.

This is not to say that there is not a sizeable body of AI programs that fall more into the category of being simply large systems programs than that of expressing heuristic algorithms. These specific sufficiently well understood applications may well be programmable with the set of primitives available in Ada.

Our conclusion that Ada is not suitable as a general AI research language should not be construed as implying that Lisp and Sail leave no room for improvement in designing languages for AI research

[1]Many of the above arguments, of course, apply as well to the general computer science research community

applications. There *is* room for improvement over existing AI languages and the opportunity to design a more effective AI programming language. We believe that the stylistic and structuring characteristics of Ada would be of value in a language for AI research, even though Ada itself cannot be that language. The AI community has resisted, and rightly so, the use of existing structured languages for AI programming because they do not provide the features and flexibility essential to AI applications. However, there is no reason to believe that either the nature of the AI applications, or the programming language features necessary for them, preclude the use of more recent concepts of programming style and structure. Indeed the more complex the program is, the more important is its structure, otherwise the intent of the program can be lost in the details of its implementation.

An AI research language should provide a powerful set of medium-level primitives in an extensible framework, with the necessary control to allow the researcher to build up the higher-level concepts relevant to his particular task, so that these higher-level constructs form a consistent and well-structured extension to the language. Lisp and Sail both provide a set of medium (and in some cases, low) level primitives in a framework allowing some degree of abstraction. While both languages contain some concept of associative or attribute access to a data base, neither language supports any notion of hierarchical construction of abstract data types, or of consistent language extension to support a user's abstractions (in the sense of the Simula CLASS structure [30]). These capabilities would enhance the ability to tailor the language to a particular problem area.

Also lacking in Lisp, and, to a lesser extent in Sail, is the ability to express within the program information either about the problem domain that is known initially or additional information that is gained during system development. In even the most experimental programs, many component data structures are well understood in advance and the structuring concepts underlying Ada are directly applicable with advantage. For other structures, less is known in advance but it is still desirable to express and enforce that which is known, and to add to this information as the program develops. An example of this would be the ability to refine the typing constraints as one learns more about the values associated with some variable or to state more general assertional constraints concerning those particular values. This capability could be used to detect inconsistencies , to achieve greater implementation efficiency and to enhance the readability of the program.

Within AI programs, as elsewhere, typing is beneficial for expressing and enforcing the domain of values over which a variable will range. For many simple variables, this domain is as well-understood in AI programs as in any other programs and strong typing is as desirable and helpful. Even where understanding is incomplete and flexibility is required, the total license of a typeless language is still inappropriate. A structure which expresses general constraints, allowing a few general operations and attributes for a wide class of objects, can then be refined to provide more detailed operations and attributes to a restricted subset of them. Possibly a highly developed version of the Simula class and subclass mechanism might meet this requirement. The Cedar project [7] at Xerox PARC, attempting to convert Mesa [26] into a language suitable for AI applications by postponing the degree of compile-time type specification required and allowing greater program interaction with the translator, linker and run-time environment, is another experimental step in this direction.

6. Features of Ada Appropriate for Re-Implementation of AI Programs

6.1. Programming Style and Structuring for Large Systems

The objectives of Ada undoubtedly differ from those of the various AI languages. Ada is intended, deliberately, to impose a highly disciplined style of programming, for use in complex projects involving large teams of programmers. In such projects it is not easy to ensure that a program is manipulated only by its original programmer even during development, let alone during maintenance over its in-service lifetime. Consequently Ada is designed to be readable and understandable rather than writable, and to minimize the cost of program maintenance. It is clear that these objectives of Ada are highly pertinent to a project to reimplement an AI program for use in an embedded system context.

Particularly important stylistic and structuring characteristics of Ada include:

- ☐ declaration of everything before use,
- ☐ strict type checking (but see Section 6.2),
- ☐ derived types and user defined types,
- ☐ packages for structuring large systems,
- ☐ use and restricted clauses to control access to non-local identifiers,
- ☐ block structure to allow declaration of local identifiers,
- ☐ lexical binding of all identifiers,
- ☐ nested control flow constructs,
- ☐ records and variant records for data structuring.

6.2. List Processing in Ada

Ada provides the language features necessary for list processing, though the language definition is equivocal about garbage collection (see Section 7.2), and without special consideration to list processing in the implementation, efficiencies may be very low.

The definition of lists in Ada could follow closely the traditions of Lisp. A list cell is a record of two components, CAR and CDR, each a list pointer. A list pointer is a record with only a variant part, whose variant discriminator, the atom flag, assumes two values, List and Atom, corresponding to the list pointer component being a list reference or an atom reference. List_Reference and Atom_Reference are access types to dynamically allocated list cells and atom cells. Only the atom cell differs significantly from Lisp in that the type checking of Ada requires that it be declared to be variant record with discrimination on the type of the atom. Garbage collection (if available) is hidden from the Ada programmer and he need not include any garbage collection flags in the list definition, though of course the actual implementation may need to conceal such flags in the list cells.

Using an Ada list is also highly similar to using a list in Lisp, though Ada draws more attention to the need to discriminate between list references and atom references and between different types of atoms. The main inconvenience to the programmer comes in the need to gather together all the various types

that might be list atoms so that the atom cell type can be defined. The atoms themselves could be set up as variant components of the atom cell, or the atom cell might contain only references to the actual atoms.

While it is unlikely that an Ada compiler designed without reference to list processing would be able to package and access list cells efficiently, the structure required in Ada is no different from that in a typical Lisp implementation and could be contained in a predefined package library associated with special optimization built into the compiler. The user could then make use of the list processing types as one would a built-in data type. As described in Section 7.2, garbage collection, beyond that to be expected of a typical Ada implementation, would also be required.

Typical type declarations for list processing in Ada might be:

```
type List_Cell, List_Pointer, List_Reference;
type Atom_Cell, Atom_Reference;

type List_Cell is
 record
   CAR,CDR: List_Pointer;
 end record;
end type;

type List_Pointer is
 record
   Atom_Flag: constant (List,Atom);
   case Atom_Flag of
   when List => LIST: List_Reference;
   when Atom => ATOM: Atom_Reference;
   end case;
 end record;
end type;


type List_Reference is access List_Cell;
type Atom_Reference is access Atom_Cell;

type Atom_Cell is
 record
   Atom_Type:constant (int,str,..);
   case Atom_Type of
   when int => i: Integer;
   when str => s: String;
   when ....
   ....
   end case;
 end record;
end type
```

6.3. Association in Ada

AI programs make use of association to represent large sparse data structures, as illustrated by the property list facilities of Interlisp and the much more extensive associative data base facilities of Sail. The use of such facilities causes of course a performance penalty in exchange for the space saving. We see no reason why a package could not be programmed in Ada to provide essentially the same associative data base facilities as are available in Sail. No modification or addition to Ada would be required. The form of expression might be held to be slightly less elegant than that in Interlisp, but would be very close to that in Sail. As was the case with list processing in Ada, it would be necessary to declare an item in the associative data base to be a variant record with a variant field for each possible item type. In addition, in order to meet the requisite efficiency considerations, it would be necessary to resort to UNSAFE PROGRAMMING within the package.

7. Crucial Deficiencies of Ada for Translation of AI Programs

7.1. Procedures as Values

As discussed in Section 3.1, AI paradigms rest heavily on the ability to use procedures as storable denotable objects. Procedures are often used as a way of representing knowledge about a particular domain. A common programming practice is to allow a variety of knowledge representations to coexist within data structures. Typically, this involves data structures where knowledge is either stated explicitly or given by a procedure which derives the necessary knowledge when called. The desirability of one knowledge representation over another may depend on the variability of the information, which in turn may be a function of the context in which the knowledge is required. The use of procedural knowledge rather than explicit knowledge could even apply to the data structure itself, where connectivity between nodes could be given procedurally rather than by explicit connection.

Procedures in Ada are neither storable nor passable objects. It is not possible to pass a procedure as a parameter to a function or a procedure, to return a procedure as the result of a function call, or to store a procedure within a data structure. The procedure falls within the category of a control structure rather than a value in a data type domain.

Clearly, the above capabilities can be simulated within Ada --at some cost; after all, one has the ability to simulate a Turing machine. The important question is whether it can be done in a clearly expressed and easily maintainable fashion.

A procedural value, as defined in a language such as Algol 68 [3], consists of an instruction pointer and that part of the environment enclosing the procedure declaration which is necessary for its execution. Thus, a procedural value is *not* just a pointer to the appropriate code segment, but also (functionally) carries with it the static environment in which it is declared. Recall that for a block-structured language allowing recursive procedures, this environment is not in general known at compile-time, since the local environment in which a procedure is declared could be a recursive instantiation of an outer procedure. One cannot therefore replace the use of a procedural value by a simple index value indicating the proper procedure to be called and a CASE statement for accessing the

denoted procedure. This works only if the environment of each use of the procedural value is identical to the environment of the procedure declaration. For this reason, it is not possible to provide an Ada package which would simulate procedural values in a transparent manner. In many cases, the program could be restructured to obtain this by moving any variables referenced by the procedure to a more global position in order that it be visible at any position from which the procedure could be called. Such a move destroys the advantages that block structure is to provide however, and can lead to obscure programming techniques.

Adding procedural variables to Ada is not a simple task. The ability to store procedures in data structures for later access leaves open the possibility that the lifetime of variables referenced from within the body of the procedure is shorter than that of the procedural variable to which the procedure is assigned. This leads to the *dangling reference* and *imposter environment* problems [25] which must be checked for in languages such as Algol 68 and Euler [11]. This check, in the absence of further language restrictions, must be performed at run-time (or more likely, ignored -- leaving open the possibility of type breaches). Further investigation would be required to derive language restrictions allowing compile-time determination of possible lifetime errors.

While adding the full capability of procedural variables may fall beyond the scope of the current Ada design, there is a very useful subcapability that could be easily added: allowing procedures to be passed as parameters.

Adding procedures as parameters to Ada does not add a redundant capability. The full power of allowing procedures to be passed as parameters is not subsumed by the generic procedure facility found in Ada. This is again due to the fact that a procedural value is characterized by its instruction pointer *together with* its necessary environment.

The use of a generic procedure with a procedure as the generic parameter achieves the same effect as the instruction component of a procedural value, and avoids the use of a CASE statement that would be necessary for accessing the procedure if an explicit index to the indicated procedure were used. It carries with it the name of the "passed" procedure (in the sense of a macro) but does not carry the environment of the procedure. Thus it cannot be used to access variables in environments that are not visible at the point of procedure call.

Even for the situations where generic procedures can suffice in place of procedures as parameters, their use may produce programs that are more difficult to read and write. A procedure that accepts a procedural parameter,such as

```
PROCEDURE apply(q:PROCEDURE(Integer) ):
 BEGIN
   x:Integer;
   compute(x);
   q(x);
 END;
```

could in principle be simulated in Ada by

```
GENERIC(PROCEDURE q(y:Integer)) PROCEDURE apply() IS
 DECLARE
  x:Integer;
 BEGIN
  compute(x);
  q(x);
 END;
```

However, for *each* call to the procedure with a different procedural parameter, it is necessary to instantiate the generic procedure by

PROCEDURE applyf_i IS NEW apply(f_i);

For a large number of possible procedures that could be passed as parameters, this becomes both lengthy to write and difficult to understand. In addition, one must rely on the language translator to realize (perhaps with the assistance of a pragma) that one is attempting to simulate a procedure as a parameter in order to bundle all generic instantiations together into one parameterized copy of the procedure. Otherwise, each generic instantiation of the generic procedure will be treated as a macro expansion, and will generate multiple copies of the procedure. Thus an efficient (and rather complex) translator will determine that one is attempting to make use of a procedure as a parameter and reconstruct its use in the implementation of the procedure. Furthermore, we feel that the source-level statement of the algorithm is cleaner and easier to understand when expresses with procedure parameters.

In our opinion, adding procedures as parameters to Ada poses no difficult implementation issues, does not introduce any possibility of type breaches if done properly[2], and would have a minimal impact on the rest of Ada. Its primary impact on the present version of Ada would be to increase the complexity of the transitive closure algorithm necessary to determine side-effects and aliasing characteristics of procedures and functions. However, restrictions on side-effects of functions and on aliasing will disappear as of the next released draft of the language [16]. Thus, in return for a slight addition in compile- and run-time[3] complexity, allowing procedures as parameters would provide an important expressive tool not just for AI programs, but also for a wide range of embedded computer applications. While this would not completely alleviate the need for procedural values during translation of AI programs, it would allow clearer expression of a significant body of AI algorithms.

[2]Procedures passed as parameters should be fully typed as in Algol 68, not as is done in Pascal.

[3]The only run-time complication should be that procedure exit may involve restoring a variable rather than fixed number of display positions.

7.2. Garbage Collection

As discussed in Section 3.1, AI programs typically operate in a domain with large numbers of alternate approaches to problem solution. In the course of investigating these alternatives, attempting to determine the best alternative to follow, a tremendous amount of storage related to each particular path may be generated. The lifetime of the generated objects is frequently tied not to the control flow of the program but rather to the duration of utility of the data (and thus is a function of the context of use). For this reason, storage for such objects must be allocated in a global *heap*, allowing storage to remain at least as long as it is referenced elsewhere in the system.

In many embedded system applications, the amount of data that must be allocated in heap storage is small, and the question of when and how to reclaim heap storage is unimportant. It may even suffice not to attempt to reclaim allocated heap space. If the designated pool of storage is sufficiently large to handle the expected number of allocations, one can avoid the expense of determining which storage is no longer needed and thus reclaimable.

Such a "no deposit, no return" approach to heap allocation is inconsistent with AI program operation. It is the case that *no* amount of initial heap allocation will be sufficient for the continued operation of many AI programs. For these programs, exhaustion of the heap is a normal and frequent facet of program operation. Thus it's not a matter of *if* the heap will be exhausted, but *when* it will happen. There must be an effective way of reclaiming obsolete storage, either by reference counts [9] or incremental tracing strategies [27], or by a full tracing of accessible objects when storage is fully exhausted.

Ada does not preclude garbage collection, but neither is it intended that Ada run-time systems will provide garbage collection capabilities. Ada provides a limited mechanism (FOR-USE) by which the user, when declaring an access type, may state the maximum number of objects that may be generated. With this information, the compiler may allocate the maximum number of objects in advance on the stack, to be deallocated when the most global access variable of that type is deallocated. For objects with limited scope of use, this provides a method of heap-type allocation with automatic reclamation. Again however, it ties allocation and deallocation to control flow (block entry and exit) rather than to data connectivity. This will not suffice for most AI applications.

What is required is either a system-provided incremental garbage collection (in the style of [17]) or a provision for the data type implementor to define a garbage collection scheme for each user-defined data type.

We wish to make a strong point that an explicit FREE operation provided to the user *is not a viable solution*. Such an approach is only marginally better than for the user to use unsafe programming to achieve storage reclamation. The premature FREEing of storage by the user leads to type breaches in the form of the dangling reference and imposter environment mentioned earlier. A FREE operation is a low-level machine dependent concept and violates the precepts of higher-level language that Ada is to provide. Embedded system programs which cannot afford to use garbage collection will also not be able to use a FREE operation as an alternative, because of the risk of heap exhaustion to which dynamic allocation exposes them. Such programs must use the FOR-USE capability.

Since garbage collection can occur only after all references to an object disappear, the logical function of a program cannot be affected by the system providing a garbage-collected finite heap as opposed to a conceptually infinite heap.[4] System performance may suffer of course. This possible degradation of performance, intolerable for many real-time applications for which Ada is targeted, is a necessary aspect of a majority of AI programs. While it can be expected that most of the DOD Ada implementations will not contain system provided garbage collection, it is *crucial* that any implementation intended to support AI programs contain such a capability. The inclusion of the garbage collector would presumably be requested via a *pragma*.

The second alternative, that of user-defined garbage collection, cannot be achieved in Ada as it is now defined. To do so would require either:

- □ that the system call a user provided subprogram when an out-of-space exception is raised. The user program would then use unsafe programming practices to do low level manipulation of the heap and associated bookkeeping,
- □ that the user provide a language-level "heap" implementation as an abstract data type. As discussed in Section 8.1, however, Ada does not currently provide the abstract data type implementor with sufficient control over the allocation and deallocation of abstract objects to perform the necessary bookkeeping information. To do so would require a more extensive initialization and finalization capability than is now possible. This can only be accomplished by allowing the user a direct Free command, leaving open the possibility that the user will free a still active cell.

7.3. More Tasking Control

At the time of writing this evaluation, the tasking features of Ada are undergoing redesign. As we have not had access to the proposed revision, our comments for the moment are quite general.

There is no reason to believe that the requirements of AI applications for parallel processing are any more demanding than those of the embedded systems applications for which Ada is intended, indeed quite the reverse. It remains to be seen whether Ada tasking control will be sufficient for embedded systems and AI applications.

A number of research projects are investigating parallel processing for programming environments, for distributed processing, or high performance computation, and are using languages with very novel tasking features. There is no reason to expect that the Ada facilities, constrained by the requirement of 'proven' techniques, should be able to express all of the programs of these research projects. However the existing proposals for Ada are particularly stylized in this area and show very little flexibility to accommodate structures other than those envisaged by the designers. In contrast for instance the proposals for the Red language show a much greater degree of flexibility, and would come much closer to meeting the requirements of the research community.

[4] This assumes that the garbage collector will always to able to reclaim sufficient storage to continue program operation

In the Appendix, Carl Hewitt of MIT provides a view of these concerns. When we have seen the revised Ada language design we will be more able to estimate the extent to which Ada will be able to meet the needs of the more demanding AI applications.

Notes on requested features (pending revised language design)

--scheduling discipline

The requirements on a scheduling discipline separate into two classes: those necessary for the logically correct operation of the system and those intended to ensure efficient allocation of a scarce resource.

Irrespective of the scheduling discipline employed, there are certain guarantees that must be provided by the implementation in order for the system to function in a logically correct manner. One such guarantee is that of *fairness*, that is, that each process demanding service will eventually receive it. The assumption of a fair scheduling policy is needed to derive *liveness*, or progress, properties of a multi-process system. These guarantees should part of the tasking specification in language definition.

In addition to guaranteeing the fairness of a scheduling discipline, performance requirements of real-time systems dictate that the user must have some capability for specifying performance aspects of the scheduling policies employed. For the allocation of the processing resource, the designers of Ada have chosen to define into the language a specific implementation mechanism rather than to simply define the desired result. The mechanism they have chosen is appropriate for some applications, particularly those involving very short tasks on a single processor, but it will be quite difficult to implement on a multi-processor. For longer tasks (greater than about 200 instructions) other scheduling mechanisms, such as deadline scheduling, are usually more effective, and can carry up to 40% more load without overloading [32].

Further, the priority scheduling mechanism defined for Ada has poor overload characteristics. It is important to distinguish between tasks that are "urgent", in that if they are to be run at all they should be run very promptly, and tasks that are "important", which may not need such rapid response but which *must* be run. To be effective under overload conditions, a scheduling strategy needs to be able to detect the presence of an overload and to know which unimportant tasks should not be processed for the duration of the overload.

Another important characteristic of a scheduling strategy for embedded systems applications is the ability to ensure that, when a low priority task blocks one of high priority, the low priority task is expedited until that of the higher priority is able to resume. Without such a feature, real-time response is difficult to ensure and consequently undesirable programming practices may be encouraged.

The common characteristic of these aspects of the scheduling discipline is that they are invisible to the programmer and have no place in the language definition. Rather they represent the strategies by which a implementation of the language achieves the best performance by the appropriate allocation of its processing resource. By explicitly predefining an implementation mechanism and by making that mechanism visible to the programmer, the precepts of higher-level language design are compromised. In addition, the particular implementation chosen for the language specification may result in a less efficient system than might have otherwise resulted.

-- process variables

Process variables are necessary for the expression of certains kinds of programs without a fixed task configurations (e.g., operating systems, distributed load balancers). Such a capability exposes the language to all of the pitfalls resulting from the use of procedure variables, namely, dangling task references, imposter task environments, type breaches, and an inconsistent run-time stack. Run-time detection of the validity of process variables will be significantly complicated by the asynchronism. For this reason, the addition of process variables should only be considered after procedure variables have been included in Ada with resolution of the resulting problems.

-- distributed computation

It is clear that Ada has no provision to support distributed computation, which is hardly surprising in that there is no agreement on what is appropriate to support it. Distributed computations could, with very great care, be written as a single Ada program. However we assume that the intention was that such systems are at present best written as a number of separate Ada programs, with any load balancing, etc., being programmed specially by the user. These issues are discussed further in the Appendix.

7.4. Resolution of Parameter Binding

Currently in Ada there are three subprogram parameter binding classes defined. These are defined in the Ada Preliminary Reference Manual (6.3) as follows:

IN The parameter acts as a local constant whose value is provided by the corresponding actual parameter.

OUT The parameter acts as a local variable whose value is assigned to the corresponding actual parameter as a result of the execution of the subprogram.

IN OUT The parameter acts as a local variable and permits access and assignment to the corresponding actual parameter.

In defining the three modes of parameter passing, there is a deliberate attempt (Rationale 7.2) to not specify a value transfer mechanism. For IN OUT parameters, for example, there is no specification of how or when the access and assignment to the corresponding actual parameter takes place. Thus, an abstract model (as well as an actual implementation) using a call-by-reference or a call-by-return-value mechanism would satisfy the IN OUT specification. In fact, any number of intermediate reflections of the value of the local formal parameter back to the corresponding actual parameter during subprogram execution would satisfy the specification. The intent of this is to allow an implementation to tailor the parameter passing strategy to the characteristics of the object being passed; e.g., depending on the size of the object one might wish to use a by-reference rather than return-value strategy. It is defined that "a program that relies on some assumption regarding the actual mechanism used for parameter passing is erroneous."

As mentioned in the Ada Rationale, the parameter transfer mechanisms exhibit a semantic difference

in programs that utilize certain combinations of aliased subprogram parameters and in the use of exception handlers to handle exceptions raised during subprogram execution. Such programs therefore are defined to be erroneous (we read this as "illegal" -- see comments on the proposed revision to Ada). In addition to these cases, passing recursively defined data structures as parameters to subprograms can lead to "erroneous" (but otherwise legal) programming practices.

Aliased subprogram parameters will in general be difficult in Ada to determine (see [2, 5] for a discussion). In the case of variables aliased (shared) between parallel tasks, it may be impossible for the translator or run-time system to prove that access of the same variable by identifiers within non-mutually exclusive tasks will occur. In the case of sharing within recursively defined data structures, *sharing is one of the major reasons for using recursive pointer domains.* To consider such uses erroneous is to eliminate much of the utility of allowing such data structures!

As mentioned, the use of exception handlers to deal with exceptions raised during subprogram execution can also lead to semantic differences between different parameter transfer mechanisms. If it happens that control is transfered to an exception handler which interrogates a variable passed as an IN OUT or OUT parameter to the subprogram, the current state will depend on the parameter mechanism employed for that variable.

This complication is acknowledged in the Rationale (7.2b), with the justification being that "the uncertainty introduced by not knowing about the implementation is of the same order as the uncertainty that already exists about the point of the exception". It is difficult to assess "the order of magnitude" of uncertainty; however, it can be pointed out that one uncertainty of exactly where the exception was raised is under programmer control and can be reduced. This is not the case for the uncertainty of when and where actual parameters are updated to reflect changes to the formal parameters. Attempts to reduce this uncertainty by interrogating the current state of global variables may result in erroneous programs; it may not be determinable from within the exception handler which variables were passed as actuals to the procedure raising the exception, and thus not determinable which variables should not be interrogated during the cleanup operations.

As the rule is currently phrased, an erroneous program can result from accessing any variable passed as an IN OUT parameter to an aborted procedure *for the remainder of the program* (at least before any subsequent assignment is made) -- since the value referenced can depend on the parameter transfer mechanism used during the procedure call. This clearly is not the desired effect.

It would seem that no generality would be sacrificed by establishing a particular parameter transfer mechanism for each binding class. Thus, for example, one could specify that the *functional* model for IN OUT parameters is based on a by-reference transfer mechanism. As this is a functional requirement, any other transfer mechanism *having the same functional effect* may be used by the translator in place of by-reference transfer. In certain situations, it may prove to be beneficial to use a by-return-value semantics in place of by-reference, having guaranteed that upon any raised exception the values of the formal parameters are copied back to the actual parameters.

Once the abstract model of parameter transfer is fixed, it becomes possible to perform effective

exception handling, and to completely define the semantics of recursive data structures passed as IN OUT parameters and of shared variables. As discussed in Section 8.1, it is also a necessary step toward allowing true abstract data types to be defined in Ada -- having paved to way toward allowing an implementation of an abstract data type to specify the semantics of assignment over the data type domain.

Note added in Revision to Evaluation: Since the time that the above analysis of the problems with the Ada parameter binding was circulated in the first draft of our evaluation, changes have been made to the definition of parameter binding [14]. In the next version of the language, the semantics of IN OUT and OUT parameter binding will be defined as a non-deterministic choice between by-reference and by-return-value parameter transfer mechanisms. The notion of an "erroneous" program then becomes purely a matter of whether the programmer anticipated such behavior in his understanding of the functioning of his program.

This definition of parameter binding is entirely consistent but not the definition that we would have chosen. The new definition guarantees that it is not possible by testing to ensure the portability of programs -- regardless of whether a given implementation makes a deterministic or non-deterministic choice for the binding of each parameter. This coupled with the elimination of aliasing and side-effects restrictions from the language places on the programmer the burden of determining whether the use of a possibly non-deterministic parameter mechanism will affect his (or her) program.

8. Desirable Features Lacking in Ada for Re-Implementation of AI Programs

8.1. True Abstract Data Type Facility

Currently, Ada provides some capability to define abstract data types, but does not actively support such efforts. While there is no explicit abstract type construction facility, it is possible to define an encapsulated data type using the PACKAGE facility. PACKAGES in Ada provide an encapsulation mechanism with which one can restrict visibility to a set of variable, type and subprogram declarations. Declarations within the PACKAGE are elaborated upon entry to the environment in which the PACKAGE is contained. Thus the PACKAGE acts as a "fence" restricting visibility to enclosed declarations without affecting their allocation scope (lifetime). PACKAGES are similar to modules in Modula [28] -- principally information hiding and data encapsulation mechanisms avoiding the complexity of abstract data type facilities such as CLU's clusters [8] or ALPHARD's forms [4].

Using the PACKAGE facility, an abstract data type as follows (using the proverbial Stack example):

```
PACKAGE fence_for_stack IS

 TYPE stack IS PRIVATE;
 FUNCTION push(x:stack;y:Integer) RETURN stack;
 FUNCTION pop(x:stack) RETURN stack;
 ...

PRIVATE

 TYPE stack IS RECORD  s: ARRAY(1..n) OF Integer :=(1..n => 0);
                       t: Integer :=1;
                END RECORD;

END;

PACKAGE BODY fence_for_stack IS

 FUNCTION push(x:stack;y:Integer) RETURN stack IS
  BEGIN ... END;
 FUNCTION pop(x:stack) RETURN stack IS
  BEGIN ... END;
 ...

 BEGIN
 ...    --Initialization code executed when package
 END;   --is elaborated

END fence_for_stack;
```

Within the scope in which the package fence_for_stack is defined, one can declare variables of type stack -- as one would with other built-in or user-defined unencapsulated data types. The representation of stack is not visible outside the package body, and thus can be modified only via the provided operations (push, pop, etc.). The above definition of stack provides for initialization of the fence_for_stack PACKAGE via the BEGIN-END clause at the end of the PACKAGE BODY and for initialization of instantiations of the stack data type via the assignments specified within the stack type declaration.

In general, a PACKAGE supports the definition of an abstract data type by providing the type declaration for the representation of the abstract data type, the declaration of any associated procedures and functions, and the declaration of auxiliary data structures common to all instantiations of the abstract data type. The auxiliary data structures contained in the PACKAGE would be used for communication and bookkeeping between instantiations -- as we explain later, this leads to a weak cousin of a "type generator" as found in Alphard.

The user cannot, however:

- □ define individual initialization of an instantiated object,
- □ provide for the updating of auxiliary structures within the PACKAGE to reflect the instantiation of abstract objects,

- ☐ define finalization of the object,
- ☐ provide type instantiation parameters,
- ☐ or have generic types.

Curiously, initialization of an instantiation is possible only in the case where the representation is in the form of a record structure. For this case, one can perform initialization of each record field by using a type declaration such as

```
TYPE abstract IS RECORD field1: T1:=e1;
                        .
                        .
                        .
                        fieldn: Tn:=en;
                    END RECORD;
```

Within the above type declaration, it is possible to specify object initialization *directly pertaining* to fields within the actual record representation. Ada defines that a type declaration is evaluated upon elaboration of the declaration, including all expressions in the right hand side of the type declaration and in any contained initialization clause. As a consequence, any functions used to initialize components of the record representation are evaluated once upon elaboration of the type declaration. Thus, one can specify only an initialization common to all instantiations of the abstract data type. This then precludes any initialization specific to a particular instantiation or to a particular context.[5]

There is also no way to update any auxiliary structures residing within the package to record object instantiation. Thus one cannot implement a type generator, responsible for generating instantiations of the abstract type and containing data structures relevant to the control and coordination of instantiations. Such data structures would be used in order to selectively initialize abstract objects and would serve to communicate between instantiations. While Ada allows such auxiliary data structures to be defined within the type manager, the only means of the appropriate updating of such structures upon object creation is to rely upon the user to call an explicit initialization routine.

Lacking also in the Ada facilities for type definition is an ability for the user to specify a final action to occur when an abstract object is deallocated. Such an action, termed *finalization*, is necessary in order to define data type which interact with their environment, either with other data structures within the type generator or in a more global context.

As an indication of this interaction, consider the problem of defining garbage collection for an abstract data type such as STRING. In order to attempt garbage collection, it is necessary to know at any given time the set of active string variables. To determine this information, the data type implementation must in some way record the allocation and deallocation of declared string variables. Such information will be used to trace the set of accessible strings when string space becomes exhausted.

[5] As an implementation consideration, it also requires that a dummy instantiation of the abstract data type be created and initialized upon elaboration of the type declaration.

Currently, the user has the capability to record the allocation of a string variable by forcing the user to call an explicit initialization of each string object before performing any other operation. However, deallocation of declared string variables occurs automatically upon scope exit, with no provision for the data type implementation to regain control in order to do final clean up and bookkeeping, and no method of enforcing any user-invoked reporting. Without this control, it is not possible for the data type implementation to maintain the information necessary to trace through the set of active strings to perform garbage collection (that is, without resorting to unsafe programming practices and interrogation of the run-time stack).

There are many additional examples of data types involving resource control, storage management, and other environmental interactions that cannot be implemented without finalization control.

Ada cannot be faulted for not including a finalization capability. Of the recent research languages Clu, Alphard, Mesa [26], Gypsy [13], and Euclid [10] designed to support data abstraction, only Euclid and the most recent specification of Alphard[6] include a finalization capability. The task of defining (and guaranteeing) finalization within Ada is far more difficult than is the case within Euclid and Alphard, due to the interaction with exception handling, dynamic storage allocation, and tasking.

We have developed [12] a language proposal and semantic model for adding finalization and arbitrary initialization to a language such as Ada. We believe it provides a consistent extension to Ada and similar languages and, in conjunction with the existing PACKAGE structure, results in a powerful abstract data type facility. The proposed facility guarantees in all conditions that each abstract object which has been created and initialized will be finalized before its scope is exited. The model applies to both statically and dynamically created abstract objects.

As mentioned early in this section, there are two other capabilities that would enhance the abstract data type facilities in Ada: type instantiation parameters and generic types. As type declarations are now defined, it is not possible to declare either compile-time parameters, through a generic type mechanism, or run-time parameters, through a type instantiation parameters.

Without these facilities, it is not possible to have user-defined type constructors or to associate attributes with user-defined types.[7] To see the desirability of such capabilities, consider the built-in Ada ARRAY_OF type constructor. The use of ARRAY_OF in a declaration such as

```
x:ARRAY(1..n,1..m) OF INTEGER
```

[6]Alphard allows a finalization clause to be attached to any object declaration, either within or outside of a FORM. This does not conform to our conception of what finalization should provide and has several difficulties which we discuss in [12].

[7]Technically, it would be possible to do this, in a roundabout manner, by declaring the package containing the type definition to be generic with the parameter being that necessary for the type declaration. One could then have a generic instantiation of the package for each desired instantiation of the contained declared type. Each such use of the type declaration would produce a *new and distinct type* however. For a type manager package containing auxiliary data structures, this would therefore not be equivalent.

accepts as parameters a type mark to be used as the element type and bounds specifications which compose the *attributes* of the array. Thus, the type constructed is a two-dimensional array of Integers with attributes 1..n for the first dimension and 1..m for the second dimension.

A type constructor with run-time attributes can not be defined by the user in Ada. Thus a user-defined type such as STRING could not define string length as an attribute (keeping in mind that we require the use of a single package to perform necessary bookkeeping), and a user-defined QUEUE_OF constructor could not be (succinctly) defined. To define type constructors as generic types and to allow type instantiation parameters, only a simple mechanism is needed. Consider the following as an example (using a user defined array type as an example):

```
GENERIC TYPE(T:TYPE) ARRAY_CONSTR(a,b,c,d:Integer) IS
   ARRAY(a..b,c..d) OF T
```

The generic instantiation

```
TYPE ARRAYINT IS NEW ARRAY_CONSTR(INTEGER);
```

would produce a new ARRAYINT type, and the declaration

```
x:ARRAYINT(1,n, 1,m)
```

would then declare x to be an instantiation of ARRAY(1..n,1..m) OF INTEGER.

The addition of these capabilities introduces compile-time complication but little additional run-time overhead. There is considerable potential gain in the flexibility of construction and use of abstract data types -- flexibility that may prove crucial for the effective use of Ada for large-scale embedded systems development.

8.2. Expanded Exception Handling Capability

Exception handling is a relatively new feature in language design, and existing AI languages provide no exception handling facilities, though the Interlisp environment does allow a form of exception interception. We do not have any examples of the use of exception handling in real AI programs, and do not feel that the Ada facility will be a serious problem during reimplementation of such programs.

Much of the experience of use of exception handling from the PL/1, Mesa, and Bliss languages indicates that abuse is as common as use and that the facility particularly lends itself to the construction of obscure programs. C.A.R.Hoare, in his evaluation of Ada exception handling facilities [15], stated the opinion that even the restricted form of exception handling present in Ada is close to the limits of the ability of programmers to construct sound systems, and that he would prefer an even simpler construct. Consequently we view with caution the quite widespread conviction amongst the AI community that they need a much more comprehensive exception handling facility.

The needs expressed for enhanced exception handling are for an ability to resume computation in the context in which the exception was generated, and for parameters to exception handlers. These two features could be provided without too much difficulty. They are both present in Mesa, although in a rather messy and ill-defined form, and D.Lomet at Yorktown Heights [22] has established the sets of

restrictions required to ensure consistency in their use. Their omission from Ada is based on a judgment as to the level of complexity appropriate to the language, and for the intended Ada user community, we have no quarrel with that judgment. Whether the AI community requires an expanded exception handling capability must be settled on the basis of future research.

8.3. **Partial Parameterization** (Contributed by Peter Hibbard)

By partial parameterization we mean the ability to supply a procedure with a subset of its parameters: a new procedure is thereby created which may subsequently be called in the normal way, by supplying the remaining parameters.

For example, if we have the procedure

```
PROCEDURE write (elem : IN integer; f : IN file_name)
```

we can create a new procedure

```
PROCEDURE writelp (elem : IN integer) IS write ( , line_printer);
```

and call it by

```
writelp (3);
```

Partial parameterization is one way of permitting knowledge to be bound to a procedure in order to affect subsequent calls of the procedure. In this section we show that the more usual way of doing this, via global variables, is inadequate for many purposes.

One particular case where these facilities are required in AI applications is where a decision procedure has a need for the service of "experts" -- procedures which are able to provide information on some area of the problem domain. Self-adapting experts start out as generalists, and they are subsequently turned into specialists as information is derived about the problem. The way in which they become experts is of no concern to the clients who use their services; also, the number of such experts and their areas of specialization are not known a priori -- as the program executes specialists are created on the fly.

Below is an analysis of the language features needed to support this paradigm. As we will show, these facilities are of use in other application areas than AI.

We intend to model experts by procedures which, when called will perform the services required of them. There are the following requirements:

- ☐ That experts can be passed to clients (as parameters) who are able to call them with parameters indicating the particular service required;
- ☐ That experts are created from generalists as the program executes, by associating with the generalists some body of knowledge which they use to perform their actions;
- ☐ That the number of experts which are needed is not known before the program starts to execute;
- ☐ That the creation of an expert in general occurs as a response to some change in the data base associated with the program, and not as a consequence of the control flow of the program;

- All the usual requirements of good program structure and good abstraction properties must be retained.

We first of all briefly review the problems of creating and manipulating procedure values. There are several formulations used in Algol-like languages to create procedure values:

1. The Ada case. The declaration of a procedure binds the environment necessary for its execution to the text of the procedure. Because the lifetime of the identifier to which the procedure value is bound is less than the lifetime of all the globally accessed identifiers, no environment retention problems can exist.

2. The Algol60 case. A procedure value may be passed as a parameter to a procedure. For the same reason as for (1), no lifetime problems exist. The only complication with respect to the execution of the program is to ensure that the necessary environment is made current when the procedure parameter is called.

We note in both these cases that the number of procedure values which is callable at any point in the elaboration of the program is statically fixed and known at compile time, since procedure values may only be called through the current bindings of the identifiers bound to procedure values. Thus the facilities above will not satisfy our requirements for an indefinite and unbounded number of procedure values existing at arbitrary points in the execution of the program.

3. Generic procedures as in Ada. These allow compile-time parametrization, effectively by textual macro substitution, this macro expansion occurring in the context of the generic definition rather than in the context of the macro call. Since generic instantiation may only be performed in a procedure declaration, generic procedures do not increase the functionality of the language beyond that present in (1), though they do add to its convenience.

4. The Algol68 case. Procedures can be manipulated in the same way as other values. It is thus possible to create an unbounded number of procedure variables (by using arrays and access variables). Note, however, that the number of different procedure values which can exist, whilst also unbounded, is determined solely by the (static) number of procedure texts, and by the current (dynamic) depth of recursive calls of procedures which contain procedure texts lexically embedded in themselves. For example there may be several instances of a procedure value derived from text P, each of which is bound to a different environment obtained by recursively calling procedure Q in which the text of P resides. Each of these instances of P may be called from the same point in the program since access paths to them may be constructed using procedure variables. Thus we have now obtained a part of the functionality required. It is unlikely, however, that this language mechanism on its own will provide us with an adequate technique for the creation of specialist procedures on the fly, for the following reasons:

   - The technique of associating specialist knowledge with a general procedure through the necessary environment of an instance of the procedure value is clumsy, error-prone, and has little relation to the abstraction we are trying to use;

   - It is improbable that the dynamic, recursive control structure required to create the experts will mesh suitably with the structure of the program which needs to make use of these experts.

We feel we can trace the cause of this problem to the following feature of algebraic block-structured languages as so far described -- that one cannot dynamically bind a value to the body of a procedure in order to cause it to have different effects when it is called. One can only place different environments at the ends of the access paths out of the procedure; this has poor abstraction properties and may only

be invoked through mechanisms unsuited to the effect it is desired to achieve. Note that this argues against using environment retention alone to obtain the required effect.

5. Partial parametrization is permitted. In this model it is possible to specialize the behaviour of a procedure by supplying it with a subset of its parameters. The procedure is not thereby called, but instead the values are bound to a new instance of the procedure value which may be subsequently called with the missing parameters.

The following example succinctly illustrates this facility, by direct application of a recurrence relation to compute the Legendre Function of degree *n*:

```
PROCEDURE legendre (n : IN integer; x : IN float) RETURN float IS
     result : float;
BEGIN
     IF n = 0 THEN
       float := 1
     ELSIF n = 1 THEN
       float :=  x
     ELSE
       float :=
         ((2*n-1) * x * legendre (n-1,x) - (n-1)
           * legendre (n-2,x)) / n
     END IF;
     RETURN float;
END legendre;
```

with calls such as

```
PROCEDURE legendre1 (x : IN float)
              RETURN float IS legendre (1, );

..... legendre1 (x) .....
```

We reject any solution to the problem of dynamically creating specialist procedures which involves methods of conveying the specializing information other than in a data object closely associated with the procedure value itself. For example,

□ Translating

```
n : integer;
get (n);
PROCEDURE legendren (x : IN float)
                RETURN float IS legendre (n, );
.....
... legendren (x) ...
```

into

```
n : integer;
get (n);
.....
... legendre (n,x) ...
```

both fails to deal adequately with the abstraction we wish to capture since we should not need to carry throughout the program the additional context to the call represented by the variable *n*, and permits accidental changes to be made to *n* between the point of reading it and using it.

□ In order to pass a specialist *P* to a procedure *Q* as a parameter to *Q*, to enable it to perform some task, we write:

```
PROCEDURE client (server : PROCEDURE (float)
            RETURN float) IS
BEGIN
      -- body which includes a call
      -- ... server (x)
END client;
```

and a call of the client:

```
client (legendren);
```

Without partial parametrization it is necessary to pass an additional parameter:

```
PROCEDURE client (server : PROCEDURE (integer; float)
            RETURN float; n : integer) IS
BEGIN
      -- same body but with call
      -- ... server (n, x)
END client;
```

and a call of the client:

```
client (legendre, n);
```

There are two objections -- first, client should not need to be aware of the means by which the server performs its task, and should not need to be aware of any additional information the server requires; second, it is unlikely that all servers which are being used by the client will have their functionality specialized in the same way: some may have their specialization done with boolean values, some with real values, etc. In this case, there is no feasible way of encoding the client unless type breaches are permitted.

A use of partially parameterized procedures that is particularly important for AI applications involves procedure variables as components of the data structures describing the objects to be processed. Partially parameterized procedures are assigned to these procedure variables to customize the processing required for each particular object.

The need to associate a value with an instance of a procedure arises frequently in all application areas. Our example shows that the more usual ways -- global variables, generic procedures, retention of environments -- are inelegant at best, and impractical in general.

9. Conclusions

It is our opinion that a useful proportion of AI programs could be reimplemented in Ada by appropriate software teams, using the original experimental version of the program as a prototype and following its algorithms, though not necessarily its detailed code. A number of relatively minor extensions to Ada would increase the proportion of AI programs that could be expressed in Ada without the use of program structures that destroy the readability or modularity of the resulting Ada program.

We have also discussed more drastic extensions to Ada that would allow the expression of almost all normal AI algorithms with relatively little modification. These extensions would have substantial interactions with other Ada language features and might conflict with other Ada objectives. We do not recommend that Ada be modified to include such extensions without further investigation.

We are particularly concerned about imprecise definition of the parameter binding in Ada; interacting with the access type, exception handling, and parallel processing features of the language to yield different results according to which parameter binding is implemented. We do not accept the definition as erroneous of any program whose results could be affected by this imprecision. For it is then extremely difficult if not impossible to determine whether any substantial program is erroneous, and consequently real systems may be expected to yield surprising results occasionally. We recognize the efficiency considerations that encouraged this dubious compromise, but we are appalled by the consequences, for any kind of programming.

We recommend that serious consideration be given to the extension of Ada to allow:

- ☐ procedures as parameters,
- ☐ a pragma requiring the presence of garbage collection,
- ☐ greater generality in type initialization,
- ☐ generic types,
- ☐ parameters to the instantiation of a type.

We do not believe that Ada will be suitable as a general research programming language for AI applications, and we do not believe that the extensions to Ada needed to meet the requirements for an AI research language are either immediately feasible or appropriate to the embedded system context for which Ada is intended.

If the advantages of modern programming language technology are desired for AI research and for the development of experimental AI applications, consideration might be given to the development of a language specifically suited to that purpose. Such a language could be part of a *programming family* surrounding Ada, attempting to maintain the basic overall structure and "flavor" of the language.

I. Ada and Multiprocessor Systems

Contributed by Carl Hewitt

It seems reasonable to expect that multiprocessor systems connected by extremely high bandwidth local networks will appear in embedded systems within the next decade. Ada has been designed for current generation hardware systems which are quite different from the new systems which are being created in advanced research laboratories. These new systems pose a number of problems which are not adequately addressed in the current Ada design. Among the most important are the following:

1. LOAD BALANCING is the ability to redistribute the computational load among the nodes on a network.
2. MIGRATION is the ability to move objects between nodes on a network. It can be used to relieve overpopulation on the nodes of a network as well as to increase efficiency by moving an object closer to where it is being actively used.
3. RECOVERY is the ability to restore an object (such as a checking account) without losing information after a crash occurs.
4. MESSAGE PASSING is the ability to communicate with objects regardless of their current location on the network.
5. REPLACEMENT is the ability to change one object into another. This capability is needed to implement continuously available systems whose objects must evolve while the system is in operation.
6. INHERITANCE provides that all objects are part of a network of descriptions in which attributes and behaviors are shared in a modular fashion. Type inheritance is needed to implement REPLACEMENT.
7. EFFICIENT REAL-TIME GARBAGE COLLECTION is the automatic reclamation of the storage of objects which are no longer accessible. Garbage collection will be needed on a network wide-basis in many future systems.

I expect that AI systems will make increasing use of parallelism in the course of the coming decade. The ability to provide the above capabilities will be crucial to any language which attempts to transfer these systems to the DoD.

Achievement of the above capabilities has a pervasive influence on the design decisions of a programming language. For example in such systems, only objects which cannot change state can be safely copied. Objects which change state cannot be safely copied because of the inconsistency which results when one machine updates one copy while another machine updates another copy. Thus the parameter passing mechanism of Ada should not provide for making copies of objects which can change state.

The greatest threat to the achievement of the above capabilities in Ada lies in the use of UNCHECKED PROGRAMMING. Instead of providing UNCHECKED PROGRAMMING, Ada should provide a small number of primitives which accomplish the goals of UNCHECKED PROGRAMMING without the effect of lowering the level of the language to essentially machine code. These primitives will allow implementations of Ada to evolve as more appropriate technology becomes available.

For example Ada should provide a FREE command which is not completely checked to recover the storage of objects. The provision of the FREE command should be regarded as a temporary engineering compromise because most current generation machines do not yet provide the micro-code support to make real-time garbage collection efficient. The FREE command is extremely dangerous in that its use can easily result in system crashes. The cause of these system crashes is often extremely difficult to isolate since it usually happens a considerable time after the offending use of the FREE command. When real-time garbage collection becomes available, programs will not have to be modified since the FREE command will be compiled as a null operation. In contrast programs using UNCHECKED PROGRAMMING will have to be extensively modified in order to use real-time garbage collection.

The original design for the programming language Mesa included the ability to make use of a "LOOPHOLE" mechanism which is very similar to the UNCHECKED PROGRAMMING in Ada. After several years of experience in using the language in programming real-time systems, the designers have decided that the loopholes must be closed in order to implement garbage collection.

II. Papers Prepared as a Result of Contract Research

The following papers have been prepared during this project:

- □ "The Suitability of Ada for Artificial Intelligence Applications", Final Report for AAG29-79-C-0216, June 1980.
- □ "Data Abstraction in Ada", submitted to the Ada Symposium, May 1980.
- □ "The Finalization Operation for Abstract Data Types", submitted to the 5th International Conference on Software Engineering, June 1980.
- □ "An Abstract Data Type Facility for Ada", to be submitted to ACM Transactions on Programming Languages and Systems.

References

[1] Ichbiah, J. D., et al. Preliminary Ada Reference Manual and Rational for the Design of the Ada Programming Language. *Sigplan Notices* 14(6), June 1979.

[2] Schwartz, R. Aliasing and Ada. Oct 1979.available as Ada-LIR.

[3] van Wijngaarden, et al, (eds.). Revised Report on the Algorithmic Language ALGOL 68. *SIGPLAN Notices* Vol. 12, No. 5, May 1977.

[4] Hilfinger, P., et al. *An Informal Definition of Alphard.* Technical Report CMU-CS-78-105, Dept of Computer Science, Carnegie-Mellon University, Feb 1978.

[5] Schwartz, R. An Axiomatic Treatment of Aliasing. Feb 1980.submitted for publication.

[6] Weinreb, D., D. Moon. *Lisp Machine Manual.* Technical Report, MIT AI Lab, Jan 1979.

[7] Bobrow, D.
personal communication.

[8] Liskov, B. et al. Abstraction Mechanisms in CLU. *Comm ACM* 20(8), Aug 1977.

[9] Berry, D., A. Sorkin. On the Time Required for Garbage Collection in Block Structured Languages. *International Journal of Computer and Information Science* (7:4), Dec 1978.

[10] Lampson, B.W. et. al. Report on the Programming Language Euclid. *Sigplan Notices* 12(2), Feb 1977.

[11] Wirth, N., H. Weber. EULER: A Generalization of ALGOL, and its Formal Definition. *Comm. of the ACM* , Jan 1966.

[12] Schwartz, R., P.M. Melliar-Smith. The Finalization Operation for Abstract Data Types. June 1980.submitted for publication.

[13] *Gypsy 2.0 Programming System 6.0* Institute for Computing Science, University of Texas at Austin, May 1979.

[14] Hilfinger, P.
private communication.

[15] Hoare, C.A.R. Subsetting of Ada, or Small is Beautiful. June 1979.

[16] Ichbiah, J.
private communication.

[17] Deutsch, P., D. Bobrow. An Efficient Incremental Automatic Garbage Collector. *Comm. of the ACM* (Vol 19, No 9), 1976.

[18] Teitelman, W.INTERLISP Reference Manual. Xerox Palo Alto Research Center, Palo Alto, 1975.

[19] Department of Defense. Ironman. Jan 1977.

[20] Church, A. The Calculi of Lambda-Conversion. *Annals of Mathematical Studies No. 6* , 1941. Princeton Univ. Press.

[21] McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* MIT Press, 1973.

[22] Lomet, D. Control Structures and the Return Statement. *Information Processing 74* , 1974.

[23] Moon, D. *MacLisp Reference Manual* Project MAC, MIT, April 1974.

[24] Wilcox, C., M. Dageforde, G. Jirak. *MAINSAIL Language Manual* Stanford University, July 1979.

[25] Chirica, L., et al. Two PARALLEL EULER Run-Time Models: The Dangling Reference, Imposter Environment, and Label Problems. In *Proceedings of a Symposium on High-Level Language Computer Architecture*, . University of Maryland, Nov 1973.

[26] Geschke, C. M. et al. Early Experience with Mesa. *Comm ACM* 20(8), Aug. 1977.

[27] Baker, H. List Processing in Real-Time on a Serial Computer. *Comm. of the ACM* (Vol 21, No 4), April 1978.

[28] Wirth, N. Modula: A Language for Modular Multiprogramming. *Software - Practice and Experience* 7, 1977.

[29] Reiser, J., (ed.). *SAIL.* Technical Report Memo AIM-289, Stanford AI Lab, August 1976.

[30] Dahl, O-J., K. Nygaard, B. Myhrhaug. *The Simula 67 Common Base Language.* Technical Report Pub. S-22, Norwegian Computing Center, 1969.

[31] Department of Defense. STEELMAN Requirements for High Order Computer Programming Languages. June 1978.

[32] Feiertag, R., K. Levitt, P. M. Melliar-Smith. *A Real-Time Operating System for Tactical Applications.* Technical Report Final Report, Project 6413, SRI International, August 1978.