June 1981

Also numbered:
PVG-19
CSL TR-208

LEVEL II

Report No. STAN-CS-81-866

12

Verifying the Absence of Common Runtime Errors in Computer Programs

BY

STEVEN M. GERMAN

Department of Computer Science

Stanford University
Stanford, CA 94305

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-81-866	2. GOVT ACCESSION NO. AD-A109433	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Verifying the Absence of Common Runtime Errors in Computer Programs		5. TYPE OF REPORT & PERIOD COVERED Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) S. M. German		8. CONTRACT OR GRANT NUMBER(s) MDA903-80-C-0159
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University (Department of Computer Sciences), Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 3423
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE June 1981 13. NUMBER OF PAGES 183
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* N/A		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE NA

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)*

N/A

18. SUPPLEMENTARY NOTES

N/A

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

Computers
Runtime Errors
Runcheck
Pascal

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

The Runcheck verifier is a working prototype system for proving the absence of runtime errors such as arithmetic overflow, array subscripting out of range, accessing an uninitialized variable, and dereferencing a null pointer. Such errors cannot be detected at compile time by most compilers. Runcheck accepts Pascal programs documented with assertions and proves that the assertions are consistent with the program and that no runtime errors can occur.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Runcheck is designed to guarantee the complete absence of runtime errors; in this respect it differs from the anomaly detection or data flow approach, which attempts to uncover runtime errors but cannot guarantee their absence. Another important distinction from previous approaches is that Runcheck is based on a detailed, rigorous semantic definition of the programming language and its data types (including pointers). Because the implementation contains a general purpose theorem prover, proofs can be arbitrarily detailed.

The thesis begins by presenting an axiomatic definition of Pascal for proving the absence of runtime errors. Our definition is similar to Hoare's axiom system, but it takes into account certain restrictions which have not been considered in previous axiomatic definitions. The definition is based on a special predicate, DEF(x), which is true if x has a properly initialized value. We discuss the problem of introducing uninitialized variables in an axiomatic definition, and construct models of the data types from nonstandard models of the integers to justify our new approach to uninitialized variables.

The thesis contains many examples of verified programs of various levels of difficulty. The verification of a four page example program is discussed in detail.

The final section draws on experience with Runcheck and the Stanford Pascal Verifier to discuss some of the major issues concerning verification and software reliability, including how verification can contribute to reliability even if absolute correctness cannot be obtained, and which applications of program verification may be feasible for large programs.

Verifying the Absence of Common Runtime Errors in Computer Programs

A thesis presented

by

Steven Mark German

to

The Division of Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Applied Mathematics

Harvard University

Cambridge, Massachusetts

June 1981

Verifying the Absence of Common Runtime Errors in Computer Programs

Steven M. German

ABSTRACT

The Runcheck verifier is a working prototype system for proving the absence of runtime errors such as arithmetic overflow, array subscripting out of range, accessing an uninitialized variable, and dereferencing a null pointer. Such errors cannot be detected at compile time by most compilers. Runcheck accepts Pascal programs documented with assertions and proves that the assertions are consistent with the program and that no runtime errors can occur.

Runcheck is designed to guarantee the complete absence of runtime errors; in this respect it differs from the anomaly detection or data flow approach, which attempts to uncover runtime errors but cannot guarantee their absence. Another important distinction from previous approaches is that Runcheck is based on a detailed, rigorous semantic definition of the programming language and its data types (including pointers). Because the implementation contains a general purpose theorem prover, proofs can be arbitrarily detailed.

The thesis begins by presenting an axiomatic definition of Pascal for proving the absence of runtime errors. Our definition is similar to Hoare's axiom system, but it takes into account certain restrictions which have not been considered in previous axiomatic definitions. The definition is based on a special predicate, DEF(x), which is true if x has a properly initialized value. We discuss the problem of introducing uninitialized variables in an axiomatic definition, and construct models of the data types from nonstandard models of the integers to justify our new approach to uninitialized variables.

The thesis contains many examples of verified programs of various levels of difficulty. The verification of a four page example program is discussed in detail.

The final section draws on experience with Runcheck and the Stanford Pascal Verifier to discuss some of the major issues concerning verification and software reliability, including how verification can contribute to reliability even if absolute correctness cannot be obtained, and which applications of program verification may be feasible for large programs.

Acknowledgements

I want to express my special appreciation to David Luckham, who invited me for a short visit to Stanford University and originally suggested the topic for this thesis. Professor Luckham generously offered to supervise my thesis research, and instead of having a short visit, I remained at Stanford until the experimental phase of the research was completed.

The members of my research committee at Harvard, Thomas Cheatham, Edmund Clarke, and Harry Lewis, gave me many helpful comments after I returned to Harvard to concentrate on writing and improving the work on formal semantics.

I also want to thank other people who offered their time to read and comment on portions of the thesis: Allen Emerson, Joseph Halpern, Dick Karp, Albert Meyer, Wolf Polak, and John Ramsdell.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA903-80-C-0159 and by the Rome Air Development Center under contract F30602-80-C-0022.

Table of Contents

Introduction

In most programming languages, there are various undefined conditions and illegal operations such as arithmetic overflow and array subscripting out of range. We call these conditions **runtime errors** because they are violations of language or implementation imposed restrictions on program execution. Current compilers do not attempt to detect runtime errors during compilation, though they commonly insert special code to test for certain errors during execution. This approach is costly in execution time and compiled program size, and of course gives no assurance that a program will run to completion.

The occurrence of a runtime error may depend on the values of data supplied to a program. For this reason, any technique for assuring the absence of runtime errors must be based on some method for **specifying** programs. Showing the absence of runtime errors is thus a natural problem in program verification.

We have been developing an automatic verifier for proving the absence of runtime errors in the language Pascal. The Runcheck system takes as input a Pascal program with entry, exit and optional invariant assertions, and proves that the specifications are consistent with the program and that no runtime errors can occur. Invariant assertions are not required in many cases because the system is able to generate simple invariants automatically, but more subtle invariants must be supplied by the user. The system currently checks for the following kinds of errors: accessing a variable that has not been assigned a value, array subscripting out of range, subrange type

error, dereferencing a NIL pointer, arithmetic overflow, division by zero, control stack overflow, exceeding heap storage bounds, and UNION type selection errors.

The language accepted by the verifier includes verifiable UNION types instead of Pascal's variant records. (Chapter 2 discusses the problems of variants and the details of our UNION types.) The verifier and our semantic definition of Pascal do not yet include REAL or SET types, but pointers are permitted.

This thesis presents an *extended* axiomatic definition of Pascal, which is the logical basis of Runcheck. The extended definition is similar to the familiar Hoare axiom system [HW73], but it takes into account certain restrictions on the computation that have not been considered in previous axiomatic language definitions.

Although the details of our semantic definition refer specifically to Pascal, most of the ideas are broadly applicable. The runtime errors which exist in Pascal are also present in many other languages, and the ideas in our semantic definition can be adopted to other languages with additional kinds of errors. ADA [Ic79] is an especially interesting case; it should be possible to define much of the language by generalizing our definition of Pascal. For instance, the problem of generalizing our definition to allow dynamic subrange types is discussed briefly in Chapter 1.

The thesis also discusses our practical experience with proving the absence of runtime errors, so that the reader can judge both the potential and limitations of this form of verification. So far, a large number of short but nontrivial programs have been verified. Chapter 3 explains in detail the complete sequence of steps followed in carrying out the verification of an interesting four page program. A list of other programs that have been verified is in the Appendix.

Obviously, the notion of runtime error does not include every kind of programming error. The runtime errors for a language are the conditions under which programs cannot continue to execute or continued execution would give undetermined results. For a program to be useful, one needs to know more about it than that it does not have runtime errors. Consider a program which is intended to copy a list made of pointers and records; it can have an error which causes it to produce the wrong result without any runtime errors in the sense we are using. Runcheck makes it possible to verify such a program at several levels of detail. For the least detailed verification, the program is submitted to Runcheck without additional specifications related to list copying. In this case, Runcheck attempts to prove only that the program is free from runtime errors. In general, it may be necessary for the user to supply some specifications and invariants even at this level of detail. For instance, the program may have a control stack overflow unless the input is acyclic. User supplied invariants would be needed in case the simple invariants generated automatically by the system are not sufficient to prove absence of runtime errors. A more detailed verification could be obtained by adding specifications saying that the result of the program is a copy of the input. An even more detailed verification could establish bounds on the performance of the program, such as the maximum number of times each statement is executed as a function of the input [LS77].

The purpose of Runcheck is to automate the routine aspects of the least detailed verifications, while still allowing the user to supply additional information for more detailed verifications. Thus although Runcheck is primarily used to perform shallow verifications, it provides a general logical framework for proving detailed properties. Every program verified by Runcheck is assured to have, as a minimum, the property that no runtime errors can occur if the entry assertion is satisfied.

Other Related Work

There has been some previous consideration of proving the absence of runtime errors in the program verification literature, but to our knowledge all previous approaches that have resulted in working implementations have been lacking in generality in comparison with Runcheck. We have both developed a general formalism for showing the absence of runtime errors and developed a working implementation. In [Si74], for instance, techniques are presented for proving absence of certain runtime errors and termination for a class of flowgraph programs, but the techniques have not been implemented. A special purpose system for checking array subscript bounds in described in [SI77]. Our system handles a wider class of runtime errors and is more general in the case of array subscripts. For example, the system described in [SI77] cannot verify correct subscripting in Example 1 of Chapter 1.

Less closely related to our work is an approach called *data flow analysis*, which has been used to detect some kinds of anomolies in programs, as in [FO76], which describes the use of data flow techniques to detect such errors as references to uninitialized variables. But there are major differences between data flow analysis and our verification approach:

1) Runcheck is based on a model of computation which is sufficiently faithful to the programming language that if the absence of runtime errors can be proven, no errors will occur during actual execution. Data flow methods obtain efficiency by using computation models which are too weak to assure absence of errors in a language as complex as Pascal. Typical data flow methods do not incorporate accurate models of complex data structures. In [FO76], arrays are treated as simple variables:

> Static data flow analysis systems such as DAVE are incapable of evaluating subscript expressions and hence cannot determine which array element is being referenced by a given subscript expression. Thus, as stated earlier, in DAVE and in many other program analysis systems arrays are treated as though they were simple variables. This avoids the problem of being unable to evaluate subscript expressions, but often causes a weakening or blurring of analytic results. As an example, consider the program shown in Figure 19. ... we see that there are two data flow anomolies present. DAVE, however, treats R as a simple variable ... and no data flow anomolies will be detected.[1]

2) The other side of the coin is that very general systems such as Runcheck cannot have a guaranteed high level of efficiency. Thus it is necessary to investigate the range of practicality of general approaches by experimenting with working implementations, as we have done. This subject is discussed further in Chapters 3 and 4.

3) Data flow techniques are usually intended to operate on the program alone without additional specifications or assertions supplied by the user. This mode of operation minimizes effort required to submit a program to the analyzer but limits flexibility and leads to greater effort and uncertainty in interpreting the results of the automatic analysis. Automatic analyzers are often unable to show absence of runtime errors without additional information from the user because i) many programs depend for their correct functioning on restrictions in their inputs, and ii) the necessary reasoning about the internal operation of programs if often too subtle without some assistance such as user supplied inductive assertions. If a program analyzer is unable for either reason to determine that a program is free from errors, the user must investigate the program further by himself to determine whether it is actually flawed. Runcheck

[1] [FO76, p. 327]

...e user a choice of either a shallow analysis requiring little user effort or a ...e thorough analysis with more effort.

Although this thesis is not primarily concerned with invariant generation, it may clarify the relationship between Runcheck and data flow analysis if we point out that sound but incomplete program analyzers can gather information for later use in a general logical framework such as the extended semantics. Data flow techniques can be used to produce a sound but incomplete analysis of a program. For instance, [CH78] is concerned with the discovery of some of the linear relations among the scalar variables in a program. Of necessity, any such analysis must be incomplete in languages as rich as Pascal, but the results are sufficient in many cases for checking errors such as array subscripting. In Runcheck, simple invariants are generated automatically by a heuristic analyzer called the documenter. This frees the user from supplying many simple invariants that are needed for proofs. The current documenter is less thorough than [CH78] for linear relations; on the other hand it deals with a broader class, producing some nonlinear relations and some assertions about array initialization. Runcheck's current heuristics are related to some of the methods previously developed by the author and described in [GW75]. We plan to investigate the possible role of data flow techniques in future versions of the documenter.

Thesis Outline

This thesis is divided into four chapters. Chapter 1 introduces the extended semantic definition of Pascal. Among the topics covered are the problems of developing an

accurate logical model of uninitialized variables, a precise definition of expression evaluation with function calls, and a practical method for verifying programs with procedural parameters. Chapter 1 concludes by discussing one of the main potential problems for the user of a verifier, the need to write detailed and repetitious assertions. We develop some simple logical properties of the extended definition which are exploited by Runcheck to reduce the need for such detailed assertions.

Chapter 2 applies the ideas of the extended semantics to a special problem in data structures: Pascal's variant records. We find that programs with variants can be handled in our semantics, but only with an undesirable restriction. At this point the discussion leaves the narrowly verification oriented point of view of Chapter 1, and proceeds to consider a range of language design, application, and implementation issues. Chapter 2 concludes by proposing new verifiable constructs to replace variants and eliminate the undesirable restrictions.

Chapter 3 presents a detailed case study of the process of verifying a moderate sized program with Runcheck. The discussion focuses on some of the strengths and weaknesses of verification as a practical tool, and attempts to convey a sense of the degree of effort required to verify programs of moderate complexity.

In Chapter 4 we present our general conclusions concerning the usefulness of verification as a tool for improving the reliability of programs.

Chapter 1. An Extended Semantic Definition of Pascal for Proving the Absence of Common Runtime Errors

The extended semantic definition of Pascal which is the logical basis of Runcheck is similar to the familiar Hoare axiom system [HW73], but it takes into account certain restrictions on the computation that have not been considered in previous axiomatic language definitions. An earlier approach to formalizing the extended semantics is presented in collaboration with D. Luckham and D. Oppen in [GLO].

Our axiomatic definition of Pascal consists of some first order theories plus axioms and inference rules for reasoning about programs. One of the first order theories concerns a predicate, DEF(x), which is true of expressions having a well defined value. The other first order theories are familiar ones such as arithmetic. Runcheck is more than a direct implementation of these logical components; practical program verifier should provide as much assistance as possible, for example, in generating inductive assertions. All of the example programs discussed in the thesis have been handled completely automatically by the system.

The theorems in the Hoare axiom system are of the form P{A}Q. Intuitively, this formula states that if P holds before executing a program A, then if and when A terminates, Q will hold. In [Ho69, HW73] and elsewhere, the relation P{A}Q is taken to be true if there is a runtime error in executing A. Hoare chose to make the interpretation that if an error occurred, the effect of the program would be "undefined," as if it had failed to terminate.

In our extended semantics, P[[A]]Q is defined to mean that if P holds, then A executes without runtime errors, and if A terminates Q will hold. Since virtually all programs are intended to execute without runtime errors, a proof of P[[A]]Q is much more useful

than one of P{A}Q, from a practical point of view.[1] If it is possible to verify the absence of runtime errors in a program, the implementation can omit the usual runtime error checking code — an increase of efficiency without loss of reliability. Also, the extended semantics is a convenient system for showing the absence of certain errors in programs that are not intended to terminate.

As is the case in other partial correctness definitions, we do not consider it an error if a program fails to terminate. The difference between our definition and others is that P[[A]]Q can hold for nonterminating A only if A is well behaved, with nothing that would *a priori* be considered a runtime error such as an arithmetic overflow, subscripting error, or control stack overflow. These specific errors are violations of the programming language; the fact of nontermination itself is not. Nevertheless, it is often desirable to be able to prove termination of a program. Proofs of termination can be carried out in a partial correctness semantics by showing the existence of bounds on the number of iterations in loops and on the depth of calls. If one wished to introduce termination as an optional part of program specifications, it would be straightforward to formalize the notion of a time bound in our logic. Since proofs of termination often require much more detail than proofs of the absence of runtime errors, one would have to decide in each case whether the additional effort to prove termination was worthwhile.

Our proof system is general purpose in that any partial correctness specification can be expressed by choosing P and Q. Absence of runtime errors is proven together with other properties. There are other possible formulations; one could develop a proof

[1] There are cases where the difficulty of proving absence of all runtime errors outweighs the additional benefit. A practical approach in such cases is to leave some errors unchecked; see Chapter 3.

system based on statements of the form SAFE[P, A], meaning that if P holds beforehand, then A executes without runtime error. The disadvantage of such a system is that proofs of the absence of runtime errors often require lemmas about more general properties of the program.

For example, consider a simple program which searches in an array A for an element equal to KEY. The elements are stored in A[1], . . . ,A[N-1]. The fast linear search stores the key in the last position of the array A before searching, so that the search loop does not have to test whether the index has become greater than N. The result of the search is returned in the variable I.

Example 1: Fast linear table search.

```
VAR N:INTEGER;
TYPE ARR=ARRAY[1:N] OF INTEGER;

PROCEDURE SEARCH(KEY:INTEGER; A:ARR; VAR I:INTEGER);
GLOBAL (N);
ENTRY DEF(N) ∧ 1≤N ∧ N≤MAXINT;

BEGIN
 A[N]:=KEY;
 I:=1;
 WHILE A[I]≠KEY DO I:=I+1;
END;
```

This program depends on the fact that A[N] has the value KEY throughout execution of the loop. Otherwise, if the key was not found in A, the loop would continue and attempt to access A[N+1], causing a subscripting error. It is necessary to prove that A[N]=KEY is an invariant of the loop, and in our extended semantics, such lemmas can be proven together in one step with the proof of absence of runtime errors.

The procedure SEARCH is presented to the Runcheck system with an ENTRY assertion stating that N has a value between 1 and MAXINT, the largest integer. The system is able in this case to verify absence of subscripting errors, arithmetic

overflow, and uninitialized variable errors (the use of the value of a variable before it has been assigned a value), automatically, given only the ENTRY assertion and program text as shown in Example 1. In particular, the necessary loop invariants including A[N]=KEY are generated automatically without any effort on the part of the user. The reader is warned not to form an opinion of the system's capabilities on the basis of this small introductory example[2] alone; a variety of more interesting programs have been handled by the system. Some of them can be found in section 7 of this chapter and in the Appendix at the end of the thesis.

[2] Note, however, that none of the three previous implementations mentioned in the Introduction, [FO76, SI77, CH78], is able to show absence of subscripting errors in this example; [CH78] does not treat relations on subscripted variables, and the implementation in [SI77] would be unable to generate the necessary invariant.

1. Chapter Outline

Chapter 1 is divided into nine sections and two appendices. Section 2 contains important definitions, particularly the definitions of the language and notation of the extended semantics. Section 3 is concerned with the predicate DEF, which is true of expressions having a well defined value. Section 4 presents some of the basic inference rules of the extended semantics. Section 5 presents a precise axiomatic definition of the evaluation of expressions in Pascal. In section 6, the definition of expression evaluation is used as the basis of a definition of Pascal statements, functions, and procedures. Section 7 develops some properties of the extended definition that are valuable when verifying actual programs. Section 8 discusses some generalizations of the extended definition, including a new method of verifying programs with procedure parameters. Following this is a discussion of our general conclusions. Finally, Appendix 1-A gives details of the implementation of the extended semantics in Runcheck, based on the principles developed in section 7, and Appendix 1-B discusses the details of a definition of simultaneous substitution for disjoint Pascal variables.

2. Preliminaries

2.1 General definitions

#T reference class (see [LS79]), used to represent the set of values of a dereferenced pointer of type ↑T.

#T⊂P⊃ value of the variable P↑ where P has type ↑T. Throughout this paper, first order language terms of the form R⊂P⊃ will denote Pascal expressions of the form P↑. Any Pascal expression involving pointers can be translated into this notation, provided that the types of the pointer variables have been specified. For further details, refer to [LS79].

POINTERSTO(#T) set of all pointer values of type ↑T.

⟨A, [I], E⟩ value of the array A after assigning the value E in the Ith position.
⟨R, .F, E⟩ value of R after R.F:=E.
⟨#T, ⊂P⊃, E⟩ value of #T after P↑:=E, where P has type ↑T.

Functions mapping Pascal expressions into types:

type(E) the type of an expression E.
indextype(A) value is R if A has type ARRAY[R] OF S.

Phrases used in a special sense:

The phrase *simple variable* is synonymous with both *variable identifier* and *declared variable*. A *selected variable* is a component of a variable identifier (e.g. A[I] is a selected variable.). A *Pascal variable* is either a variable identifier or a selected variable [JW75].

Simultaneous Substitution for Identifiers.

If P(X) is a formula where X = [x1, . . . ,xn] is an ordered set of free variable identifiers, then P(A), where A = [a1, . . . ,an] is an ordered set of terms, stands for the result of simultaneously substituting the ai for the xi in P.
If the set X of free variable identifiers of a formula P(X) is partitioned into subsets X1 and X2, then P(X1, X2) stands for P(X), and P(A1, A2) where A1 and A2 are ordered sets of terms, stands for the result of simultaneously substituting in P the terms in A1 for the variables X1 and the terms in A2 for the variables X2.

Substitution for a Pascal Variable.

$P|^{v}_{t}$ **where v is any term denoting a Pascal variable, is defined recursively as follows.**

$P|^{x}_{t}$ **where x is an identifier, stands for P with t substituted for x.**

$$P|^{v[i]}_{t} = P|^{v}_{\langle v,[i],t\rangle}$$

$$P|^{v.f}_{t} = P|^{v}_{\langle v,.f,t\rangle}$$

$$P|^{v\subset p\supset}_{t} = P|^{v}_{\langle v,\subset p\supset,t\rangle}$$

2.2 Disjoint Pascal Variables

Intuitively, two Pascal variables are disjoint iff an assignment to one of them cannot affect the value of the other. It is obvious that in languages with array subscripting and pointers, disjointness is a dynamic property — it depends on the values of variables. For instance, A[i] and A[j] are disjoint iff $i \neq j$.

If $v1, \ldots, vn$ are disjoint Pascal variables, it is possible to define the simultaneous substitution

$$P|^{v1 \quad \ldots \quad vn}_{t1 \quad \ldots \quad tn}$$

of n expressions for n Pascal variables, in terms of the sequential substitutions defined above in 2.1. This definition and the formal definition of disjointness are needed only for the procedure call rules; details are presented in Appendix 1-B.

2.3 Formulas in the extended semantics

The syntax of formulas is ordinary, and is included here mainly for reference. A formula1 is a pure first order formula. The syntactic category of program statements includes all executable Pascal statements plus some additional statements which are used only at intermediate steps during proofs. The new statement types, known as *evaluation statements* and *assume statements*, do not initially appear in programs, but can be introduced by certain rules during the course of a proof. Evaluation statements correspond to the action of evaluating an expression or computing the location of a variable. Assume statements are used by some of the proof rules to record previously justified logical assumptions at points within the body of an executable program.

Implicitly associated with each formula is a set of declarations of constants, variables, types, and defined procedures and functions, corresponding to a static scope in a program. The syntactic distinction between declared and undeclared symbols is made with respect to the scope. It is assumed that all name conflicts in the scope are removed by renaming. Also, for readability, we will feel free throughout the thesis to omit parentheses whenever the formula can be determined from operator precedence.

<variable>::= <declared variable> | <undeclared variable>

<op>::= <Pascal built in function>
| <declared function sign>
| <undeclared function sign>

<term>::= <variable> | <constant> | <op> (<termlist>)
| (<term> <infix arithmetic operator> <term>)

<termlist>::= [<term> [, <term>]*]

<predicate>::= <declared boolean function sign>
| <Pascal built in predicate (=, ≠, <, ≤)>
| <undeclared predicate sign>

<atomic>::= <predicate> (<termlist>) | True | False

<formula1>::= (<formula1> <logical connective> <formula1>) | ¬ <formula1>
| ∀ <undeclared variable> (<formula1>)
| <atomic>

<statement>::= <Pascal executable statement>
| <assume statement>
| <evaluation statement>
| <statement>; <statement>

<assume statement>::= ASSUME <formula1>

<evaluation statement>::= Eval <Pascal expression>
| Locate <Pascal variable>

<subprogram declaration>::= <Pascal function declaration>
| <Pascal procedure declaration>

<formula of unextended definition>::= <formula1>
| <formula1> {<statement>} <formula1>
| <formula1> {<subprogram declaration>} <formula1>

<formula>::= <formula1>
| <formula1> [[<statement>]] <formula1>
| <formula1> [[<subprogram declaration>]] <formula1>

Throughout the paper, we will distinguish between the type of an expression and its sort in the many sorted first order language. By the type of an expression, we mean its Pascal type according to the scope. By the sort of an expression, we mean its sort in the first order language. Except for subranges, the sort of an expression is the same as its type. Integer and integer subrange expressions are of sort integer. Similarly, expressions whose type is a subrange of an enumerated type have the same sort as the enumeration. A sort will be said to *cover* both the type with the same name and all subranges of the type.

To be well formed, a statement must satisfy the syntax and type requirements of the programming language [JW75]. Because of the correspondence between types and sorts, an expression satisfies the type requirements of the programming language iff it

is a well formed term according to the sorts. A formula1 is a first order formula which may contain free occurrences of declared and undeclared variables. Each term or atomic formula whose outer sign is declared or Pascal predefined, must also satisfy the type requirements of the programming language.

2.4 Notation for the extended semantics

The axioms and inference rules in the extended semantic definition are actually schemes, or infinite sets of axioms and rules; in this respect, our system is no different from previous axiomatic definitions. When a scheme is applied, information from the program scope must be substituted in certain places. To specify the information that is to be substituted, we use a meta notation. An expression involving a function or predicate sign in ***Bold Italics*** indicates a term or formula to be substituted. Instances of the axiom or rule are formed by evaluating the italicized meta expression to produce a term or formula. For example, the rule for assignment to a whole variable is:

$$\frac{P\ [\![\text{Eval } y]\!]\ \textbf{\textit{Inrange}}(y, \textbf{\textit{type}}(x)) \wedge Q\Big|_{y}^{x}}{P\ [\![x := y]\!]\ Q}$$

Consider a typical context:

```
TYPE s=1..500;
VAR g:s; h:INTEGER;
. . .
g := h+4;
```

Since g is a subrange variable, the assignment statement will cause a subrange error unless h+4 is in the correct range. ***Inrange***(y, ***type***(x)) is the notation for a formula which asserts that the value of y is in the range of the variable x. In the context of

the example, the desired instance of the rule is:

$$\frac{P\ [\![Eval\ h+4]\!]\ 1 \le h+4 \wedge h+4 \le 500 \wedge Q\Big|^{g}_{h+4}}{P\ [\![g := h+4]\!]\ Q}$$

2.5 Formula Constructing Functions

Inrange(<expression>, <type>)

Inrange is a function mapping <expression> × <type> → <formula>. The expression must be of a sort which covers the type.

if type is a subrange a..b,
Inrange(expression, type) → a≤expression ∧ expression≤b.

otherwise,
Inrange(expression, type) → TRUE.

Disjoint(<Pascal variable>, <Pascal variable>)

The function Disjoint maps a pair of Pascal variables into a formula which is true iff the variables are disjoint. Refer to Appendix 1-B for a detailed definition of Disjoint.

Disjoint-set(<set of Pascal variables>)

For any finite set of Pascal variables, Disjoint-set constructs a formula which is true iff all pairs of variables in the set are disjoint.

3. Theory of definedness: the predicate DEF

There are a number of possible ways to include the concept of an uninitialized variable in a programming language definition. What we need is some way to keep track of which variables have been assigned well defined values at any point during execution of a program. For the moment, let us restrict attention to simple integer program variables. We will be considering the two related questions:

> What mathematical model should be used to represent the values of and operations on integer variables which can be uninitialized?
>
> What first order axioms will be used to prove statements about the model?

As an initial model of definedness, it is natural to assume the existence of a single undefined value Ω not contained in the set of integers $\mathbb{Z}$, and to let integer program variables range over the extended domain $\mathbb{Z}' = \mathbb{Z} \cup \{\Omega\}$. In such a model, we can show that a program never uses the value of an uninitialized variable by showing that whenever a value is accessed, the value is not equal to Ω. Thus we can assign the predicate DEF(X) its intended meaning, "X has a defined value," by defining $\mathrm{DEF}(X) \equiv X \neq \Omega$ in this model.

A somewhat subtle point, to which we will return later, is that it is not necessary in the model to assume variables have the initial value Ω if we want to prove absence of uninitialized variable errors. When we develop the semantics of executable statements, our approach will be to make no assumptions about initial values: a variable may start out having any value in the domain. However, we consider a program free from errors in accessing a variable only if we can prove that in all executions, the value accessed is not equal to Ω. The only way for a variable to

become restricted to be unequal to Ω is for it to be assigned a defined value. Thus we can prove absence of uninitialized variable errors without making any assumption about the initial values of variables.

A problem arises if we try to formulate a first order theory of the domain $\mathbb{Z}'$. Since arithmetic operations such as + and · must be extended to total functions on $\mathbb{Z}'$, we have to choose interpretations for terms such as $0 \cdot \Omega$ and $1 + \Omega$. It is not hard to see that no matter what extension is chosen, a domain with only one nonstandard element cannot satisfy the familiar theory of arithmetic on the integers. Letting $\Omega+1 = \Omega$ in the model would invalidate the sentence $\forall x\ x+1 \neq x$, while if we let $\Omega+1 = n$, for some integer constant n, it would would follow that Ω was an integer in $\mathbb{Z}$. In fact, it is well known that nonstandard models of first order Peano arithmetic must have at least a countably infinite number of nonstandard elements (this is discussed in logic texts, for example, [BM77, En72]). We can retain a domain with one undefined element only by adopting an unconventional theory of arithmetic containing sentences such as $\forall x\ (DEF(x) \supset x+1 \neq x)$ instead of $\forall x\ x+1 \neq x$. Since all integer calculations in such a theory would be cluttered with references to DEF, we will choose to modify the initial approach by using a larger domain to retain the familiar theory.

Our intended model of definedness for integer variables is now the following: let $\mathcal{B}^*$ be a nonstandard model of arithmetic with domain $\mathbb{Z}^*$. Then define $DEF^{\mathcal{B}^*}(X)$ to be true for the standard integers and false elsewhere in $\mathbb{Z}^*$.

We now turn attention to the first order theories involved. Let L_Z be the first order language of the theory of arithmetic. With no loss of generality, choose L_Z such that it does not contain the symbol DEF. (The reason for this choice will become apparent shortly.) Let $\Sigma_Z \subseteq L_Z$ be some "reasonable" set of axioms for integer arithmetic. Also

choose standard theories Σ_E for enumerated sorts and Σ_{DS} for the assignment, selection, and extension operations on complex data structures. We will need the notion of equality of compound data objects (DS1); other details of a theory of data structures can be found in [LS79].

DS1a) if x and y are expressions of a record sort, and f1, . . . ,fn are the field names,
x=y ≡ (x.f1=y.f1 ∧ . . . ∧ x.fn=y.fn).

DS1b) if x and y are expressions of sort ARRAY[a . . b] OF t,
x=y ≡ (∀i a≤i≤b ⊃ x[i]=y[i]).

DS1c) if s and t are reference classes of the same sort,
s=t ≡ POINTERSTO(s)=POINTERSTO(t) ∧ (∀p∈POINTERSTO(s) s⊂p⊃ ≡ t⊂p⊃)

We now list the axioms Σ_{DEF} of the theory of DEF.

DEF1) for every constant c, DEF(c) is an axiom.

DEF2) if e is of an enumerated sort (c1, . . . ,cn),
DEF(e) ⊃ e=c1∨ . . . ∨e=cn.

DEF3a) DEF(a)∧DEF(b) ⊃ DEF(a ⊕ b)
where ⊕ is an operator in {+, −, *, =, ≠, <, ≤, AND, OR, NOT}

DEF3b) DEF(a)∧DEF(b)∧b≠0 ⊃ DEF(a/b)∧DEF(a DIV b)∧DEF(a MOD b)

DEF4a) if x is an expression of sort ARRAY[a . . b] OF t,
DEF(x) ≡ (∀i a≤i∧i≤b ⊃ DEF(x[i])).

DEF4b) if r is of a Pascal record sort, and f1, . . . ,fn are the record field names,
DEF(r) ≡ DEF(r.f1)∧ . . . ∧DEF(r.fn).

DEF4c) if #t is of a reference class sort,
DEF(#t) ≡ (∀p∈POINTERSTO(#t) (p≠NIL ⊃ DEF(#t⊂p⊃)).

The resulting theory of DEF is still not logically complete, e.g. because it does not say much about the undefined values. But we should not expect to find such details in a programming language definition. All of the properties needed for proving absence of errors in programs have been included.

As the final step in introducing DEF, we will look at a many sorted model of each

sort under the combined axioms $\Sigma = \Sigma_Z \cup \Sigma_{DEF} \cup \Sigma_E \cup \Sigma_{DS}$, and show why the theories are satisfied in the intended models.

Integers and Integer Subranges. Recall that integers and integer subranges have the same sort in the first order logic. Therefore, subrange restrictions are not expressed in the first order logic; they are introduced only in the logic of programs. The model $\mathcal{I}^*$ applies to both integer and integ r subrange variables.

Because we chose L_Z not to contain the symbol DEF, the intended interpretation $DEF^{\mathcal{I}^*}(x)$ trivially satisfies Σ_Z. The other axioms for the integer sort are DEF1, DEF3, and in the case of integer components in compound data structures, DEF4 and Σ_{DS}. DEF1 is satisfied because $DEF^{\mathcal{I}^*}(x)$ is true for standard integers. Observe that DEF3 is satisfied because $\mathbb{Z}$ is closed under all arithmetic operations. The remaining cases of pointers and compound data structures are explained in later sections.

Let us now see what could have gone wrong if DEF had been included in L_Z. We wanted our treatment of DEF to work with any reasonable Σ_Z; this freed us from the problem of choosing a particular theory of arithmetic. One reasonable component of Σ_Z is an axiom scheme for Peano induction. For simplicity, let us consider a scheme for induction or the natural numbers; with trivial changes, our comments will apply to the integers.

$$\Phi(0) \wedge \forall n\, (\Phi(n) \supset \Phi(n+1)) \supset \forall x\, \Phi(x). \qquad (P)$$

Instances of the axiom scheme are formed by substituting a formula of L_Z for $\Phi(x)$ in P. In particular, if DEF(x) was a formula in L_Z, we would have $DEF(0) \wedge \forall x\, (DEF(x) \supset DEF(x+1)) \supset \forall x\, DEF(x)$ in Σ_Z. From DEF1 and DEF3, it

would be possible to deduce $\forall x$ DEF(x), contradicting the intended interpretation, in which there are values for which DEF(x) is false. Obviously, the predicate DEF would be of no use if its only interpretation was true for all values; we avoided this difficulty by assuring that DEF was not part of the theory of arithmetic.

One final point is that there are many models of the integer sort under the axioms Σ. No first order theory uniquely defines the mathematical notion "X is an integer," and so no matter what set of axioms we supply, there will be models in which DEF is true for nonstandard values. Because the axioms in Σ_{DEF} do not require DEF to be false for any value, one nonstandard interpretation is DEF(x) = True. The existence of nonstandard interpretations does not detract in any way from our use of DEF to prove absence of runtime errors. Since $\mathfrak{I}^*$ with $DEF^{\mathfrak{I}^*}$ is a model of the integer sort under Σ, theorems derived from Σ are true statements about $\mathfrak{I}^*$ and $DEF^{\mathfrak{I}^*}$.

Enumerated Sorts. Let TYPE en = (c1, . . . ,cn). Then a model $\mathfrak{E}$ for the sort en can be defined by $|\mathfrak{E}| = \mathbb{Z}^*$, $ci^{\mathfrak{E}} = i$, and $DEF^{\mathfrak{E}}(x) = 1 \le x \le n$. The standard operations and relations on en are defined in the obvious way. A point of caution is that we must not have x=c1 ∨ . . . ∨ x= cn in the theory of an enumerated sort — this would lead to the problem with DEF(x) = True. Instead, we have axiom DEF2, which permits values which are not DEF. Note that the same model is used for a subrange of en.

Pointers. A model of pointers must deal with two kinds of objects: pointer values and reference classes or sets of dynamic variables. In this section we assume familiarity with pointer semantics as presented in [LS79]; our purpose is to show how to model the defined and undefined values in a way which satisfies Σ_{DEF} and reasonable choices of Σ_{DS}.

For pointer sorts without the pf operation, the only interpreted symbols are NIL and DEF. For any pointer sort we will assign the structure $\mathfrak{P}$ = ($|\mathfrak{P}|$, $NIL^{\mathfrak{P}}$, $DEF^{\mathfrak{P}}$) with $|\mathfrak{P}| = \mathbb{Z}^*$, $NIL^{\mathfrak{P}} = 0$ and $DEF^{\mathfrak{P}} = DEF^{\mathfrak{B}^*}$. Note that we use a single construction even for recursive pointer types. Now consider an arbitrary pointer definition TYPE ptr = ↑t. Using $\mathfrak{P}$ and the preceeding sections, there is a model for the sort t under the combined axioms Σ if t is a simple sort. Let us assume for the moment that given sorts t1, . . . ,tn, a many sorted model $\mathfrak{T}_i$ of Σ for each ti, and a scalar type s, that we can form a many sorted model $\mathfrak{A}$[s; $\mathfrak{T}$i] for the sort ARRAY [s] OF ti, and a model $\mathfrak{R}$[f1 . . . ; fn; $\mathfrak{T}$1; . . . ; $\mathfrak{T}$n] for the sort RECORD f1:t1; . . . ; fn:tn END. Therefore, let us assume in general that there is a many sorted model $\mathfrak{T}$ of the sort t. Then by constructing the reference class, we can form a many sorted model for all of the operations on sort ptr.

In our model, a reference class consists of two components, a function mapping $\mathbb{Z}^*$ into $|\mathfrak{T}|$, and an integer indicating the number of dynamic variables which have been created. To insure that equality between reference classes depends only on the values of the currently existing dynamic variables, we assign a single value $t0 \in |\mathfrak{T}|$ to all members of the reference class outside of the currently existing variables.

More formally, a reference class is an ordered pair $(m: \mathbb{Z}^* \to |\mathfrak{T}|,\ n \in \mathbb{Z})$ such that $n \geq 0$ and $\forall x\ ((x<1 \vee x>n) \supset m(x) = t0)$.

We now assign interpretations to the predicates and functions on pointers and reference classes. Let r = (m,n) be a reference class and p a pointer.

P1) $r\langle p\rangle \leftrightarrow m(p)$

P2) $POINTERSTO(r) \leftrightarrow \{i \mid 0 \le i \le n\}$

P3) $\langle r, \langle p\rangle, e\rangle \leftrightarrow (m', n)$
where $m'(p) = e$ and $m'(q) = m(q)$ for $q \neq p$.

P4) $r \cup \{q\} \leftrightarrow (m, n+1)$
provided $q \leftrightarrow n+1$.

P5) $DEF(r) \leftrightarrow \forall i\ (1 \le i \le n \supset DEF(m(i)))$

Notes: P2) POINTERSTO(r) is the set of all pointers to dynamic variables which have been allocated in reference class r. P4) The *extension* operator $r \cup \{q\}$ represents the result of allocating a new dynamic variable in r; q is a new pointer of type ptr which points to the new dynamic variable. Later in this chapter we will use extension to define the Pascal NEW procedure.

The reader can easily check that this interpretation satisfies the standard properties of pointers and reference classes and that DS1 and Σ_{DEF} are also satisfied.

Remark: the theory of reference classes in [LS79] is weak; it does not include an induction principle for reasoning about non-constant sequences of pointer operations. If we had a stronger theory of data structures, $\Sigma_{DS+} \supset \Sigma_{DS}$, how would the interpretations of reference classes and DEF be affected? To answer this question, we have to delimit the class of reasonable theories of data structures. If we omit the interpretation of DEF for reference classes and consider the intended interpretations of data structures, axiom DEF4c will define the relation DEF on reference classes of variables of sort t to be $DEF(r) \equiv \forall x\ (1 \le x \le n \supset DEF(m(x)))$ where $r = (m, n)$. A definition of this form is consistent in a reasonable choice of Σ_{DS+} even with induction; furthermore, DEF on reference classes of variables of sort t will be the trivially true relation iff DEF is trivially true on sort t. As long as Σ_{DS+} is chosen so

that 1) it is satisfied by the intended interpretation of the theory of data structures without DEF, and 2) DEF is not trivially true for sort t in the intended interpretation, then the complete axiom system will have a model in which DEF has the desired meaning on reference classes.

Arrays and Records. The construction of a domain of array values for the theory of sort ARRAY [a . . b] OF t is analogous to the construction of the set of reference classes of dynamic variables of sort t. The array values are triples $(m: \mathbb{Z}^* \rightarrow |\mathfrak{T}|,\ n1 \in \mathbb{Z},\ n2 \in \mathbb{Z})$ where n1 and n2 are integer values corresponding to the index bounds a and b. Record values in the model are elements of the direct product of the domains corresponding to each of the record components. Selection is interpreted in the obvious way.

As in the case of reference classes, DEF is assured to have the desired meaning if Σ_{DS} contains a reasonable inductive theory.

3.1 The relationship between DEF and Inrange

In Pascal, every subrange type is bounded by two constants,[3] a . . b. Thus according to the definition of Inrange, Inrange(x, s) implies DEF(x), if s is a subrange. This follows from the properties of the ≤ ordering of the integers. For example, it is a theorem in the theories of integer ordering and DEF that $\forall x\ ((1 \le x \wedge x \le 4) \supset DEF(x))$, because the standard properties of integer ordering imply that

$$\forall x\ ((1 \le x \wedge x \le 4) \supset (x=1 \vee x=2 \vee x=3 \vee x=4))$$

[3] More flexible languages are discussed in section 8.

and each of these constants is DEF. Note, however, that

$$\forall x \forall y \forall z \, (DEF(x) \wedge DEF(z) \wedge x \leq y \wedge y \leq z) \supset DEF(y) \qquad (3.1)$$

is *not* a theorem in the combined axiom system; it cannot be proven by induction on the integers because Σ_χ does not contain any instances involving DEF. In fact, there are nonstandard interpretations of the theories of DEF and integers for which formula 3.1 is not satisfied.

Also note that it is not necessary for a variable to be Inrange if it is DEF: under the axioms of DEF, there can be a variable of a declared subrange type, whose value is both DEF and not Inrange. In the definition of P [[A]] Q, no program is permitted to assign a value to a subrange variable unless the value is Inrange. If P [[A]] holds, a subrange variable can be out of bounds only before it has been assigned a value.

4. Fundamental inference rules.

The following two rules are included in both the unextended and extended definitions.

Concatenation of programs. (CONCAT)

$$\frac{P\ \{A\}\ Q,\quad Q\ \{B\}\ R}{P\ \{A;\ B\}\ R} \qquad \frac{P\ [\![A]\!]\ Q,\quad Q\ [\![B]\!]\ R}{P\ [\![A;\ B]\!]\ R}$$

Consequence rule. (CONSEQ)

$$\frac{P \supset Q,\ Q\ \{A\}\ R,\ R \supset S}{P\ \{A\}\ S} \qquad \frac{P \supset Q,\ Q\ [\![A]\!]\ R,\ R \supset S}{P\ [\![A]\!]\ S}$$

These rules will be used implicitly, beginning in the next section on the semantics of expression evaluation. Later, after $P\ [\![A]\!]\ Q$ has been defined, we will develop its logical relationship to $P\ \{A\}\ Q$ in more detail.

5. Expression Evaluation.

This section introduces and defines *evaluation statements*. Evaluation statements have the forms

Eval <Pascal expression>
Locate <Pascal variable>

and in the extended semantics, they can be combined with Pascal statements and assertion statements to form the general statements which appear inside brackets in a formula P [[A]] Q. Evaluation statements will be used in section 6 to define the conditions for error free execution of Pascal statements containing expressions and variables.

The statement Eval E corresponds to the action of evaluating the expression E, which may not have side effects. P [[Eval E]] Q is defined to mean that if P holds then E evaluates without runtime error, and if E terminates then Q will hold. Since E does not have side effects, P and Q refer to states with the same values for variables. By having two assertions, it is possible to make partial correctness statements about function calls. For instance, if f is a (strictly) partial function,

$$P(x)\ [[\text{Eval } f(x)]]\ Q(x, f(x))$$

may be a provably true statement about the evaluation of f(x), while the pure first order statement

$$P(x) \supset Q(x, f(x))$$

would not be true since it does not account for divergence of f(x).

The other form of evaluation statement, Locate V, corresponds to the action of computing the location of a variable. The difference between this and evaluating a

variable is that to compute a location, all of the subscripts must be evaluated and all dereferenced pointers must be evaluated, but the variable itself need not have a value. For instance, to execute the assignment statement A[j]:=e, the subscript j must have a value in the correct range, but the left hand side A[j] is not required to have a value. The definition of A[j]:=e is expressed in terms of Locate A[j], and Eval e, since the right hand side must yield a value. The formula P [[Locate V]] Q is defined to mean that if P is true, then the location of V can be computed without execution errors, and if the computation terminates, Q will hold.

The exact meaning of expression evaluation is often a point of confusion in programming languages and definitions. The definitions presented here assume that sufficient restrictions are used to prevent side effects. Pascal [JW75] assumes a fixed order of evaluation within statements and expressions, so the final value of an expression is well determined even in the presence of side effects. It is a simple matter to replace a function definition which has side effects by an equivalent procedure definition, by adding a new VAR parameter to return the function value. Thus it is possible to rewrite a Pascal program in which functions have side effects into an equivalent program in which function calls are replaced by procedure calls and all expressions are free of side effects. This transformation would convert the evaluation of an expression with side effects into a sequence of procedure calls involving some new variables to store temporary values. Since this transformation can be easily mechanized, our Pascal semantics are indirectly applicable even to programs with function side effects.

If runtime errors are not being considered, as in the original Hoare axiom system, function calls without side effects can be defined by the following rule,

$$\frac{I_f(X1,\ldots,Xn,G)\ \{\text{Function } f(X1{:}t1;\ldots;Xn{:}tn){:}t_f;\ B\}\ O_f(X1,\ldots,Xn,G),\quad P\ \{\text{Eval } A1;\ldots;\text{Eval } An\}\ I_f(A1,\ldots,An,G)\wedge(O_f(A1,\ldots,An,G)\supset Q)}{P\ \{\text{Eval } f(A1,\ldots,An)\}\ Q} \qquad \text{(F1)}$$

which states that evaluation of f(A1, . . . ,An) can be reduced to the evaluation of A1, . . . ,An in order, followed by the application of f, if I_f and O_f are shown to be valid entry and exit assertions for f. G is the set of read only global variables, and B is the body of the function f.

A fine point to be considered at the practical level is that some compilers change the order of evaluation within expressions if there are no side effects. If the evaluation of an expression terminates, it terminates with the same result under all orderings. Since the truth of P {Eval E} Q depends only on whether evaluation of E terminates and the value of each subexpression, all orders of evaluation are equivalent with respect to P {Eval E} Q. The truth of P {Eval E} Q can be determined by choosing any possible ordering and considering whether it is true for that ordering. Rule F1 above, depends on choosing one ordering. Thus F1 is correct even if there is reordering.

The situation is different when proving absence of runtime errors. Then, different possible orders of evaluation must be considered separately. For instance, an expression such as f(x)+a[i] might have a runtime error if i is out of range. If f(x) is evaluated first and does not terminate, the error cannot occur. But if the order is changed and a[i] is evaluated first, the error could occur. Since different orders of evaluation can give different results, we define P [[Eval E]] Q to be true iff every order of execution is error free and Q will hold after every terminating execution.

Another complication is the possibility of *short circuit* evaluation in Boolean expressions. In evaluating an expression such as r AND s, when the value of r is False,

Pascal permits compilers to omit the evaluation of s. The expression r AND s is assumed to have the value False because r is False. Observe that if s does not terminate or if it has a runtime error, the short circuit has a different partial correctness semantics from full evaluation. For example,

P [[Eval r AND s]] False

may be true for full evaluation but not for short circuit. Short circuit evaluation is really a form of branching within expressions. The axiomatic definition assumes that full evaluation is used. Some languages, such as ADA, permit short circuit evaluation in certain contexts but require the user to explicitly request it. This seems to be a cleaner approach, and we show below (rule E3S) how it can be formalized in the extended semantics.

In summary, our detailed semantic definition of Pascal statements will be based on partial correctness assertions about evaluation of expressions and variables. It is argued that even in the absence of side effects, the definition of expression evaluation should as a practical matter account for possible variations in the order of evaluation. We will give an axiomatic definition that does not assume any fixed ordering. On the other hand, function call rule F1 can be used if evaluation order is fixed, or if runtime errors are not considered.

The rules defining P [[Eval e]] Q are as follows:

Expression evaluation.

$$\frac{P\ [\![\text{Locate } V]\!]\ DEF(V) \wedge Q}{P\ [\![\text{Eval } V]\!]\ Q} \qquad \text{(E1)}$$

(V is any Pascal variable.)

P [[Eval A]] Q

P [[Eval (⊗ A)]] Q (E2)
(where ⊗ is one of the monadic operators, +, -, NOT)

The following rule for evaluation of an operator expression contains three conditions. The first two assert that A and B evaluate without runtime error if P holds. These conditions make the rule independent of any fixed order of evaluation, by requiring either operand to evaluate correctly if evaluated first. The third condition states that after both operands have been evaluated, Q must hold. Since there are no side effects and the first two conditions have established that the operands evaluate without errors, the order in the third condition is not significant. Notice, though, that the first condition is redundant because the third one also requires A to evaluate safely. In stating the rest of the rules, we will omit redundant conditions such as this.

P [[Eval A]] True,
P [[Eval B]] True,
P [[Eval A; Eval B]] Q

P [[Eval A⊗B]] Q (E3)
(where ⊗ is a relation sign or boolean connective.)

Rule E3S formalizes evaluation of ADA conditions. In ADA, the boolean conditions for controlling IF and WHILE statements etc. can have one of the forms

<expression> AND THEN <expression>
<expression> OR ELSE <expression>

which indicate that the left hand expression is to be evaluated first, after which the right hand expression will be evaluated only if its value is needed to determine the value of the condition. The following rule for evaluation of A AND THEN B states that it must always be possible to evaluate A, and that 1) if A is false, Q must hold, and 2) if A is true, it must be possible to evaluate B, after which Q must hold.

P [[Eval A]] ¬A ⊃ Q,
P [[Eval A; ASSUME A; Eval B]] Q

P [[Eval A AND THEN B]] Q (E3S)

Maxint is an undeclared integer variable representing the range on which integer arithmetic operators do not overflow. The axiomatic definition makes no assumption about the values of Maxint. In order to prove absence of overflow, the user must supply assertions relating Maxint to the computations in the program.

P [[Eval B]] True,
P [[Eval A; Eval B]] -MAXINT≤A⊕B≤MAXINT ∧ Q

P [[Eval A⊕B]] Q (E4)
(where ⊕ is one of the arithmetic operators, +, -, *)

P [[Eval B]] True,
P [[Eval A; Eval B]] B≠0 ∧ Q

P [[Eval A⊕B]] Q (E5)
(where ⊕ is DIV, MOD, or /)

Maxint can have any value such that integer arithmetic does not overflow in the range -Maxint . . Maxint. Note that many computers use twos complement arithmetic, in which the smallest negative integer has an absolute value one greater than the largest positive integer. This situation (and other possible number systems with asymmetrical ranges) can be more accurately modeled by introducing a separate variable Minint to stand for the smallest integer, and making the obvious changes in rules E2, E4, and E5.

The following rule defines the evaluation of a function call f(A1, . . . ,An), where each of the Ai is a value parameter and G is a list of read only global variables. For error free evaluation of the function call, each of the Ai must evaluate and yield a value in the proper range. The second the third premises of the rule state that if I_f and O_f are valid entry and exit assertions for f, then they can be used to show P[[Eval f(A)]]Q.

If the parameters A and G satisfy the entry condition I_f, then O_f will hold on exit. Also, f(A,G) will be DEF and Inrange -- these properties are assured by the declaration rule.

for i=1, . . . ,n, P [[Eval Ai]] ***Inrange***(Ai, ti),
I_f(X1, . . . ,Xn,G) {Function f(X1:t1; . . . ;Xn:tn):t_f; B} O_f(X1, . . . ,Xn,G),
P [[Eval A1; . . . ;Eval An]] I_f(A,G) ∧ (O_f(A,G) ∧ DEF(f(A,G)) ∧ ***Inrange***(f(A,G), t_f) ⊃ Q)
---------- (E6)
P [[Eval f(A1, . . . ,An)]] Q

Location Validity.

P [[Locate V]] P (L1)
(this is an axiom for any declared variable identifier V)

P [[Locate R]] Q
---------- (L2)
P [[Locate R.F]] Q
(where R is of a record type with a .F field)

P [[Eval Z]] Z≠NIL ∧ Q
---------- (L3)
P [[Locate Z↑]] Q
(where Z is of a pointer type)

P [[Eval I]] True,
P [[Locate A; Eval I]] ***Inrange***(I, ***Indextype***(A)) ∧ Q
---------- (L4)
P [[Locate A[I]]] Q
(where A is of an array type)

Example 2: Show Q [[Eval a[i]+p↑]] True, where

Q ≡ DEF(i) ∧ 0≤i≤100 ∧ DEF(a[i]) ∧ 0≤a[i]≤25 ∧ DEF(p) ∧ p≠NIL ∧ p↑=6 ∧ 1000≤MAXINT

with the variable declarations
VAR a: ARRAY[0:100] OF INTEGER;
VAR i: INTEGER;
VAR p: ↑INTEGER;

By applying the inference rules in reverse, we can find simpler sufficient conditions for the formula to be true. We will continue to work backwards until we reach

sufficient conditions that are obviously true. At this point, the formula will be proven, because it will be possible to construct a formal proof by starting with the final conditions and applying the inference rules until the original formula is deduced. The first step is to use rule E4 in reverse, reducing the problem of proving a statement about Eval a[i]+p↑ to proving statements about Eval a[i] and Eval p↑.

Q [[Eval p↑]] True, (5.1)
and Q [[Eval a[i]; Eval p↑]] -MAXINT ≤ a[i]+p↑ ≤ MAXINT. (5.2)

Before finishing the example, we pause to mention a fact about the extended semantics which is helpful in removing redundancy from proofs. Since expressions do not have side effects, we can assume in proofs that the state does not change when an expression is evaluated. The following lemma states this fact in a useful form.

Lemma. ⊢ P [[Eval e]] True, iff ⊢ P [[Eval e]] P.
⊢ P [[Locate e]] True, iff ⊢ P [[Locate e]] P.

Another point about redundancy is that when applying the inference rules directly to prove P [[Eval E]] Q, the proof of error free execution of some subexpressions may appear many times. A mechanical evaluator of the preconditions can easily take the repetition into account and only verify each subexpression once.

Continuing the example, show 5.1:

Q [[Eval p↑]] True

← Q [[Locate p↑]] DEF(p↑) (by E1)

← Q [[Eval p]] p≠NIL ∧ DEF(p↑) (by L3)

← Q [[Locate p]] DEF(p) ∧ p≠NIL ∧ DEF(p↑) (by E1)

← Q ⊃ (DEF(p) ∧ p≠NIL ∧ DEF(p↑)) (by L1 and CONSEQ)

← True. (by definition of Q)

Next, show Q [[Eval a[i]]] True

← Q [[Locate a[i]]] DEF(a[i]) (by E1)

← Q [[Eval i]] DEF(a[i]),
and Q [[Locate A; Eval i]] 0≤i≤100 ∧ DEF(a[i]) (by L4)

These last two formulas are trivially provable, since the assertion Q implies that i has a value, and the whole variable A is always a valid location by L1. Having shown that both a[i] and p↑ evaluate without any errors, we can use the CONCAT rule to infer that one can be evaluated after the other, i.e.

Q {Eval a[i]; Eval p↑} True (by CONCAT). (5.3)

It only remains to show that there is no overflow, formula 5.2.

Q {Eval a[i]; Eval p↑} -MAXINT ≤ a[i]+p↑ ≤ MAXINT

← Q ⊃ -MAXINT ≤ a[i]+p↑ ≤ MAXINT
(by CONSEQ and lemma applied to 5.3)

← True.

Example 3: User defined partial functions in expressions.

```
VAR x: INTEGER;
VAR a: ARRAY[0:100] OF BOOLEAN;

FUNCTION sqrt(n: INTEGER): INTEGER;
  ENTRY True;
  EXIT  0≤sqrt≤n
  BEGIN
     % if n < 0, then loop forever without execution errors;
       otherwise, set sqrt ← integer part of square root n.
     %
   ....
  END;
```

Suppose the function sqrt has been defined to correctly return the integer square root of n unless n is negative, in which case it loops forever without runtime errors.

Using the function declaration rule which will be given in section 6.3, it is possible to prove

True [[Function sqrt(n:INTEGER):INTEGER; body]] 0≤sqrt(n)≤n. (5.4)

The entry and exit specifications of sqrt can then be used to show that if sqrt is called with an argument x whose value is less than 100, the location of the variable a[sqrt(x)] can be computed without runtime error.

DEF(x) ∧ x≤100 [[Locate a[sqrt(x)]]] True

← DEF(x) ∧ x≤100 [[Eval sqrt(x)]] True, (5.5)

and DEF(x) ∧ x≤100 [[Locate a; Eval sqrt(x)]] 0≤sqrt(x)≤100 (by L4) (5.6)

Using the function call rule E6, the first part 5.5 reduces to

DEF(x) ∧ x≤100 [[Eval sqrt(x)]] True

← DEF(x) ∧ x≤100 [[Eval x]] True,

and True [[Function sqrt(n:INTEGER): INTEGER; body]] 0≤sqrt(x)≤x,

and DEF(x) ∧ x≤100 [[Eval x]] True ∧ (0≤sqrt(x)≤x ∧ DEF(sqrt(x)) ⊃ True)

which are all true.

The second part 5.6 can be simplified

DEF(x) ∧ x≤100 [[Locate a; Eval sqrt(x)]] 0≤sqrt(x)≤100

← DEF(x) ∧ x≤100 [[Eval sqrt(x)]] 0≤sqrt(x)≤100 (by L1 and CONCAT)

← DEF(x) ∧ x≤100 [[Eval x]] (0≤sqrt(x)≤x ∧ DEF(sqrt(x)) ⊃ 0≤sqrt(x)≤100)
(by E6)

← DEF(x) ∧ x≤100
[[Locate x]] DEF(x) ∧ (0≤sqrt(x)≤x ∧ DEF(sqrt(x)) ⊃ 0≤sqrt(x)≤100)
(by E1)

← DEF(x) ∧ x≤100 ⊃ DEF(x) ∧ (0≤sqrt(x)≤x ∧ DEF(sqrt(x)) ⊃ 0≤sqrt(x)≤100)
(by L1 and CONSEQ)

← True

6. Extended axiomatic semantics of Pascal

6.1 Assume statements

The meaning of the statement ASSUME L is that L can be assumed to be a true assertion at a certain point in a program. Assume statements do not initially appear in programs, but can be introduced during the course of a proof to record logical assumptions which hold at points within a program. For instance, the rule for IF statements reduces a formula involving IF L THEN S1 ELSE S2 to two formulas for the two cases of the condition L. In one formula, the statement ASSUME L records the assumption that L was true, and in the other formula, ASSUME ¬L records the assumption that L was false.

$$\frac{(P \wedge L) \supset Q}{P\ [\![\text{ASSUME } L]\!]\ Q} \qquad \text{(ASSUME)}$$

6.2 Executable statements

Assignment statements

The following rule applies to all assignment statements.

$$\frac{P\ [\![\text{Eval e}]\!]\ \text{True},\quad P\ [\![\text{Locate pv; Eval e}]\!]\ \textit{Inrange}(e, \textit{type}(pv)) \wedge Q\Big|_{e}^{pv}}{P\ [\![pv := e]\!]\ Q} \qquad \text{(ASSIGN)}$$

where pv is any Pascal variable

In order for P [[pv := e]] Q to hold, it is necessary for the assignment to execute without any runtime errors, and for Q to be true in the updated state. The rule

requires the right hand side, e, to evaluate without runtime error and to yield an initialized value; the location calculation for left hand side pv is also required to be free from errors. If pv is a subrange variable, the Inrange clause requires the value of e to be in the correct range. The updated formula Q is formed by substituting e for the Pascal variable pv, using the definition of substitution given in section 2.1.

IF statements

P [[Eval L; ASSUME L; S1]] Q,
P [[Eval L; ASSUME ¬L; S2]] Q

P [[IF L THEN S1 ELSE S2]] Q (IF)

CASE statements

for i=1, . . . ,n, P [[Eval X; ASSUME X=C_i; S_i]] Q,
P [[Eval X]] X∈{C_1, . . . ,C_n}

P [[CASE X OF C_1:S_1; . . . ;C_n:S_n]] Q (CASE)

The C_i are lists of constants for each branch of the CASE statement. The second condition requires the CASE expression X to evaluate to one of the constants in one of the C_i.

NEW procedure

The following axiom states that the effect of the Pascal statement NEW(x), where x is a variable identifier of a pointer type, is to change the value of x to a new pointer value xo, and to add the new value xo to the reference class.

$$\neg(x0 \in \text{POINTERSTO}(\#T)) \wedge \text{DEF}(x0) \wedge x0 \neq \text{NIL} \supset Q\Big|^{\#T}_{\#T \cup \{x0\}}\Big|^{x}_{x0} [\![\text{NEW}(x)]\!]\, Q(\text{NEW1})$$

where x is an identifier of type ↑T (pointer to object of type T),
x0 is a fresh identifier not appearing in Q,
#T is the reference class for type T,
#T ∪ {x0} stands for the reference class after adding an object pointed to by x0.

The antecedents on the left side of the rule state that 1) the value x0 generated by NEW is a new pointer, not a pointer to the reference class #T, 2) x0 has an initialized value, and 3) x0 is not the pointer NIL. The term #T ∪ {x0} represents the new reference class after the dynamic variable x0↑ has been allocated. A more complete discussion of POINTERSTO and the operation of adding new elements to a reference class can be found in [LS79].

The following rule reduces a NEW statement involving a selected variable to a NEW statement with an argument which is an identifier.

$$\frac{P\ [\![\text{NEW}(S0);\ S:=S0]\!]\ Q}{P\ [\![\text{NEW}(S)]\!]\ Q} \qquad \text{(NEW2)}$$

where S0 is a new identifier not appearing in the scope, P, or Q.
the declaration VAR S0: *type*(S) is added to the scope.

WHILE statements

$P \supset I$,
I [[Eval B; ASSUME B; S]] I,
I [[Eval B]] $\neg B \supset Q$

P [[INVARIANT I WHILE B DO S]] Q (WHILE1)

In this rule, the invariant is chosen to be true before each evaluation of the While test B. The rule can be rearranged to correspond to other choices of invariants.

6.3 Functions and procedures

6.3.1 Function declaration

With the function declaration rule, one can infer that I and O are valid entry exit specifications for a function f, if for inputs satisfying I, the body of the function executes without runtime errors and assigns a final value to the function which satisfies the exit assertion O.

$I(X1, \ldots, Xn, G) \wedge DEF(X1) \wedge \ldots \wedge DEF(Xn) \wedge \mathbf{Inrange}(X1, t1) \wedge \ldots \wedge \mathbf{Inrange}(Xn, tn)$
[[B]] $O(f, X1, \ldots, Xn, G) \wedge DEF(f) \wedge \mathbf{Inrange}(f, t_f)$

$I(X1, \ldots, Xn, G)$ [[Function $f(X1:t1; \ldots; Xn:tn):t_f$; B]] $O(f(X1, \ldots, Xn), X1, \ldots, Xn, G)$ (FD)

where f has the function declaration
FUNCTION $f(X1:t1; \ldots; Xn:tn):t_f$;
GLOBAL G;
ENTRY $I(X1, \ldots, Xn, G)$;
EXIT $O(f, X1, \ldots, Xn, G)$;
B;

The rule requires that the function have only value parameters $X1, \ldots, Xn$ and a set of read only globals G. The rule assumes that each of the value parameters has an initialized value in the correct range; this assumption is justified by the call rule,

which checks the actual parameters. If global variables are accessed, the entry assertion must assert that they have been initialized.

In the exit assertion O(f,X1, . . . ,Xn), the variable f stands for the value returned by the function. The rule checks that the body assigns f a value in the correct range. As we will see in section 7.4, the condition ***Inrange***(f, t_f) appearing after execution of the body is redundant. Because the declaration rule requires f to be DEF after execution of the body, it is not necessary to require f to be Inrange.

6.3.2 Note on Global Variables

Runcheck requires the user to declare lists of all global variables that could potentially be accessed or altered by each subprogram. The system checks the lists by a syntactic examination of the subprogram body. For instance, a global variable g which is used in an assignment statement g := e, must be declared read write. Also, if the body of p contains calls to q, then all globals listed for q must be listed for p.

Reference classes are a special case of global variables which are implicitly accessed or altered although they do not appear explicitly in the executable program text. If a subprogram evaluates p↑, this is considered an implicit access to a reference class. An assignment p↑ := e is considered an implicit write to the reference class. The system requires all reference classes which are used as globals of a subprogram to be explicitly listed by the user as global parameters.

The presence of a pointer formal parameter does not necessarily imply that a reference class will be accessed or altered by a subprogram. For instance, a procedure p with a VAR formal parameter x which is a pointer to an integer,

```
TYPE ptr = ↑INTEGER;

PROCEDURE p(VAR x: ptr);
BEGIN x := NIL END;
```

may assign to x without altering the reference class #INTEGER. No globals would be listed for this procedure, since it changes only the pointer x and not any of the integer variables pointed to.

On the other hand, in a procedure p2 which assigns to x↑, it would be necessary to list the reference class #INTEGER as a read write global,

```
TYPE ptr = ↑INTEGER;

PROCEDURE p2(VAR x: ptr);
GLOBAL (VAR #INTEGER);
BEGIN x↑ := 0 END;
```

because an integer variable accessed by a pointer is changed.

Observe that depending on the actual argument, a call to the procedure p above could have the effect of changing a reference class, as in the call

```
TYPE ptr = ↑INTEGER;
     ptr2 = ↑ptr;
VAR y: ptr2;

p(y↑);          % changes #ptr %
```

which changes the reference class #ptr of variables of type ptr which are accessed by pointers. In this case #ptr is not considered a global, although the call rules do account for the fact that part of #ptr is altered by being passed as a VAR parameter. Which reference class is altered in this example depends on the call, not on the definition of p. For example, in the call

```
TYPE ptr = ↑INTEGER;
    ptrarray = ARRAY[1..100] OF ptr;
    ptrptrarray = ↑ptrarray;
VAR z: ptrptrarray:

p(z↑[50]);
```

z is a pointer to variables of type ptrarray, z↑ is an array of pointer variables, and z↑[50] is a pointer to an integer, and hence the correct type to be an argument to procedure p. The variable which p changes in this case is an element of an array accessed by a pointer, and this causes a change to the reference class #ptrarray.

The ability of a procedure with a VAR pointer parameter to change different reference classes depending on the actual parameter, is exactly analogous to the ability of a procedure with a VAR integer parameter to change components of different integer arrays.

```
PROCEDURE q(VAR x: INTEGER);
BEGIN x:= 0 END;                    % no globals %
```

The first call in

```
TYPE arr = ARRAY[1..500] OF INTEGER;
VAR v1, v2: arr;

q(v1[50]);
q(v2[50]);
```

alters part of v1, but the second one alters part of v2.

6.3.3 Procedure declaration

$$\frac{I(X,Y,G) \wedge DEF(X1) \wedge \ldots \wedge DEF(Xm) \wedge \mathit{Inrange}(X1,t1) \wedge \ldots \wedge \mathit{Inrange}(Xm,tm) \quad [\![B]\!]\ O(X,Y,G)}{I(X,Y,G)\ [\![\text{Procedure p(X1:t1; . . . ;Xm:tm; VAR Y1:u1; . . . ; VAR Yn:un); B}]\!]\ O(X,Y,G)} \quad (PD)$$

where p has the procedure declaration
PROCEDURE p(X1:t1; . . . ;Xm:tm; VAR Y1:u1; . . . ; VAR Yn:un);
GLOBAL GR, VAR GW;
ENTRY I(X,Y,G);
EXIT O(X,Y,G);
B;
GR are the readonly global variables,
GW are the read write global variables,
G stands for the set of all global variables, GR ∪ GW.

Like the function declaration rule, the procedure declaration rule assumes that the value parameters are initialized by each call with values in the correct range. On the other hand, there is nothing unusual about procedures that work correctly with uninitialized VAR parameters. Consider a simple procedure p which is called with an integer j and two array variables, x and y, and assigns x[j] the value y[j].

```
TYPE s = 1..100;
TYPE arr = ARRAY[s] OF INTEGER;

PROCEDURE p(j: s; VAR x, y: arr);
BEGIN
  x[j] := y[j];
END;
```

Since the procedure does not test the range of j before executing the assignment, a call to p will produce a subscripting error unless j is between 1 and 100. Also, the actual variable supplied for y[j] must have been assigned a value before the call to p. No other restrictions are needed to assure error free execution. In particular, p will work regardless of whether x has been initialized, and regardless of whether portions of y other than y[j] have been initialized. For instance, the following sequence executes without errors.

```
VAR a, b: arr;
VAR k: INTEGER;

BEGIN
  k := 50;
  b[k] := 1000;
  p(k, a, b);
          % now a[50] = 1000 %
END;
```

The behavior of p can be specified by providing it with entry and exit assertions.

```
TYPE s = 1..100,
TYPE arr = ARRAY[s] OF INTEGER;

PROCEDURE p(j: s; VAR x, y: arr);
INITIAL y = y0;
ENTRY DEF(y[j]);
EXIT y = y0 ∧ x[j] = y[j];
BEGIN
  x[j] := y[j];
END;
```

The entry assertion states that y[j] has a value when p is called. Note that since j is a value parameter with a subrange type, the declaration rule assumes that it will be supplied with a value in the correct range — this will be checked by the call rule. The Initial statement simply introduces a new name y0 to stand for the initial value of y at the time of entry to the procedure. The exit assertion states that the value of y is unchanged, and that x[j] is equal to y[j].

To summarize the point of this example, all of the rules for subprograms assume that value parameters must be supplied with initialized values in the correct range. This is our interpretation of what it means to correctly call a subprogram with a value parameter. No such assumption can be made for VAR parameters, and so it is necessary to describe the behavior of each one by means of entry and exit assertions.

It is of course possible for there to be implementations of Pascal, in which calls with

value parameters will produce the desired results in some cases even if the actual parameter is not fully initialized. This is merely an artifact of certain possible implementation techniques. Our definition attempts to capture what is meant by the language itself, and is intended to be sufficiently restrictive to be consistent with all possible implementations.

As was mentioned earlier, the initial value of local variables is not specified by the function or procedure declaration rules. Another approach, which seems reasonable at first glance, is to assert that every local is initially undefined. This is not needed in the extended semantics, because for P [[A]] Q to be valid, every variable must be assigned a value which is DEF before its value is used.

The declaration rules could be modified to specify an initial value for locals, but this would unnecessarily complicate the definition and lead to confusion in applying the extended semantics. It would be possible to introduce a new constant C_S for each sort to stand for the initial value. The axioms would be changed to state that for each of these constants, $\neg DEF(C_S)$, and also $\neg DEF(t)$ for terms t formed by selecting components of C_S. For each local L, $L=C_S$ would be added as a premise in the declaration rule. But this is an unnecessary complication. Also, it does not accurately model the implementation of Pascal, in which initial values are left unspecified to reduce overhead. For this reason, it would give confusing results in practice. If a program, A, never used two variables of the same sort, x and y, and otherwise executed without errors, it would be possible to prove that the variables were equal after the program,

$$P \{A\} x=y.$$

Such a result differs from the implementation and probably conceals a programming error.

6.3.4 Procedure call

The procedure call rule requires each value parameter to evaluate without runtime error, yielding a value in the correct range, and each VAR parameter to yield a location without runtime error.

for i=1, . . . ,m, P [[Eval Ai]] ***Inrange***(Ai, ti),

for i=1, . . . ,n, P [[Locate Vi]] True,

I(X,Y,G) [[Procedure p(X1:t1; . . . ;Xm:tm; VAR Y1:u1; . . . ; VAR Yn:un); B]] O(X,Y,G),

P [[Eval A1; . . . ;Eval Am; Locate V1; . . . ;Locate Vn]] ***Disjoint-set***(V ∪ G) ∧ I(A,V,G)

$$\wedge\ \forall Z,GB\ (O(A,Z,GR,GB) \supset Q\left|\begin{matrix} V1 & \cdots & Vn & GW1 & \cdots & GWk \\ Z1 & \cdots & Zn & GB1 & \cdots & GBk \end{matrix}\right.) \qquad \text{(PC1)}$$

P [[p(A1, . . . ,Am,V1, . . . ,Vn)]] Q

Each of the actual VAR parameters, Vi, must be a distinct Pascal variable not in GW. Note that this definition depends on the definition of substitution when Vi is not an identifier.

7. Metatheory of the extended definition

In this section, we discuss some properties of the extended definition which are helpful in reducing the complexity of program specifications and the length of proofs.

By itself, the extended semantics is not a complete solution to the problem of verifying the absence of common errors. In practice, there are two main kinds of difficulty in doing actual verifications. These practical difficulties were carefully considered in the design of the Runcheck system.

The problem of redundancy in proofs is solved in Runcheck by a special simplifier which efficiently eliminates redundant verification conditions.

A more serious problem is the need for lengthly, detailed specifications and inductive assertions in programs. Several distinct approaches are needed to deal with this problem. In Appendix 1-A, we discuss the derived WHILE rule, which shows how the extended definition reduces the need for detailed documentation. The derived WHILE rule and other rules are logically justified by certain simple properties of the theory of the extended definition, which are presented in the remainder of this section.

7.1 Ordinary Semantics Lemma

Any specification for an executable statement A which is provable in the extended definition is also provable in the ordinary definition (this does not apply if A is a subprogram declaration).

Lemma 7.1 If $\vdash P\ [\![A]\!]\ Q$, then $\vdash P\ \{A\}\ Q$.

The significance of this lemma is that all specifications, even those involving DEF, are theorems of the ordinary system.[4] The extended definition only places more restrictions on the allowable computations. Consistency of the extended definition is a direct consequence of this lemma.

7.2 Specification lemma

When proving complicated specifications for a program, it is sometimes helpful to prove the specifications without considering possible runtime errors, and then prove separately that no errors occur. In this way, the details about runtime errors can be isolated in the proof. The next lemma says that proofs in the extended definition can always be factored in this manner.

Lemma 7.2 If $\vdash P \{A\} Q$, and $\vdash P1 [\![A]\!] Q1$, then $\vdash P \wedge P1 [\![A]\!] Q \wedge Q1$.

The reason for this is that if both $P \{A\} Q$, and $P1 [\![A]\!] Q1$ can be proven separately, then it is always possible to combine the proofs to show $P \wedge P1 [\![A]\!] Q \wedge Q1$.

The design of the automatic Documenter in Runcheck is based on this lemma. The documenter constructs inductive assertions[5] that are valid in the ordinary semantics. The assertions can then be assumed true in proofs in the extended semantics. Thus the documenter does not have to consider possible runtime errors while constructing the invariants.

[4] In the case of built in procedures, it is necessary to choose slightly nonstandard definitions if the resulting system is to be complete with respect to specifications involving DEF. The "ordinary" system that we have in mind has axioms stating that the results of built in procedures such as READ and NEW are DEF.

[5] Refer to [Ge78] for details of the documenter

7.3 LESSDEF lemma

One of the basic properties of the extended definition is that if P [[S]] Q holds, S cannot assign an uninitialized value to any variable. Over any sequence of statements that executes without runtime error, the extent of variable initialization cannot decrease.

LESSDEF(x, y), a predicate for two terms of the same sort, is defined to be true if y is at least as completely initialized as x.

LD1) if x and y are of the same simple sort,
LESSDEF(x, y) ≡ DEF(x)⊃DEF(y).

LD2) if x and y are of the same record sort, and the field names are f1, . . . ,fn,
LESSDEF(x, y) ≡ LESSDEF(x.f1, y.f1)∧ . . . ∧LESSDEF(x.fn, y.fn).

LD3) if x and y are of sort ARRAY[a..b] OF t,
LESSDEF(x, y) ≡ (∀j a≤j≤b ⊃ LESSDEF(x[j], y[j])).

LD4) if x and y are of sort REFCLASS(t) for some t,
LESSDEF(x, y) ≡ (∀p∈POINTERSTO(x) LESSDEF(x⊂p⊃, y⊂p⊃)).

The LESSDEF lemma says that for any variable in a program that executes without errors, the final value will be at least as fully initialized as the initial value.

Lemma 7.3 If ⊢ P [[A]] True, and v is a declared variable identifier then,

$$\vdash P \wedge v' = v \; [[A]] \; \text{LESSDEF}(v', v)$$

where v' is a new identifier not appearing in P, A, or the scope.

In Runcheck, the lemma is used to reduce the need for detailed assertions on loops and procedures. If a variable is known to be DEF before entering a loop, it is not necessary to state in the invariant that it continues to be DEF. Similar assertions about VAR parameters can be omitted from procedure specifications.

Example 4: Merging two sorted arrays

This example shows how Runcheck uses the Lessdef lemma to reduce the need for repetitious, detailed assertions. The program takes as input previously sorted arrays A and B of length 100 and merges their contents into the array C, which has length 200. The user has supplied only an ENTRY assertion saying that A and B are fully initialized, and an EXIT assertion saying that C is fully initialized. The interesting aspect of this example is that the initialization of C takes place in two loops. The first loop partially initializes C, merging elements from A and B until either A or B has been completely transferred. Then the initialization of C continues in either the second loop or the third loop.

```
TYPE INARR=ARRAY[1:100] OF INTEGER;
TYPE OUTARR=ARRAY[1:200] OF INTEGER;
VAR I,J,N:INTEGER;
VAR A,B:INARR; C:OUTARR;
ENTRY DEF(A)∧DEF(B);
EXIT DEF(C);
BEGIN
N:=100;
I:=1;
J:=1;
INVARIANT DEFRANGE(1, I+J-2, C)
     ∧ 1≤I ∧ I≤N+1 ∧ 1≤J ∧ J≤N+1
WHILE (I≤N) AND (J≤N) DO
     BEGIN
     IF A[I]≤B[J] THEN BEGIN C[I+J-1]:=A[I]; I:=I+1 END
          ELSE BEGIN C[I+J-1]:=B[J]; J:=J+1 END;
     END;
I'←I;
INVARIANT DEFRANGE(I'+N, I+N-1, C) ∧ I'≤I ∧ I≤N+1
WHILE I≤N DO BEGIN C[I+N]:=A[I]; I:=I+1 END;
J'←J;
INVARIANT DEFRANGE(J'+N, J+N-1, C) ∧ J'≤J ∧ J≤N+1
WHILE J≤N DO BEGIN C[J+N]:=B[J]; J:=J+1 END;
END
```

The system will verify

$$DEF(A) \wedge DEF(B) \; [\![body]\!] \; DEF(C)$$

i.e., that the program does not have any execution errors and that no elements of C are missed. All of the other variables are initialized before the first loop. Still, it is necessary to prove that they are DEF each time they are accessed. In the case of a variable such as I, Runcheck uses the Lessdef lemma to infer that it has a value everywhere in the program after the assignment I:=1. Even though I is changed on the first loop, it is not necessary to write DEF(I) (or A, B, J, N) as an invariant.

In many array programs, the arrays are either supplied as fully initialized parameters, or are initialized at the beginning. Without the Lessdef lemma, it would be necessary to have invariants repeating the fact that an array or other data structure is DEF at various points within a program.

Consider now the more complicated case of proving DEF(C). The system automatically generates the statements shown in ***bold italics***. By examining the first loop, one can see that at any time, values have been assigned to the positions C[1], . . . ,C[I+J-2]. This fact is discovered by the system and is expressed in the invariant as

DEFRANGE(1, I+J-2, C).

DEFRANGE is a special predicate used to express that a subrange of an array is DEF. Its definition is

DEFRANGE(x,y,a) ↔ (∀i x≤i≤y ⊃ DEF(a[i])).

The invariant for the second loop states that C[I'+N], . . . ,C[I+N-1] are DEF, where I' stands for the value of I before entering the second loop. Similarly, the assertion for the third loop states that C[J'+N], . . . ,C[J+N-1] have been assigned values. The system also produces the arithmetic inequalities shown on each loop.

To be able to prove the exit assertion, DEF(C), it is necessary to show that all of C[1], . . . ,C[200] have values after the third loop. Notice that each invariant only describes the initializations done by its own loop. For instance, the third invariant only deals with the last part of C, and does not repeat the fact that the first part of C is initialized by the first loop. Runcheck uses the Lessdef lemma to infer that the first part of C continues to be DEF, even though that fact is not included in the later invariants. Thus the invariants shown are sufficient to prove that C is fully initailized on exit. The documenter's assertions are also sufficient to show that the program executes safely.

7.4 Inrange lemma

The Inrange lemma says that a program for which P [[A]] True holds cannot cause the value of a subrange variable to become out of range (when started in a state which satisfies P). If a subrange variable is known to always be DEF at some point in a program that executes without errors, then the variable must be Inrange at that point. To begin, we define Inrange*, a formula constructor similar to Inrange. The difference between the two is that Inrange asserts that a subrange variable is in the correct range and is always true for other types, while Inrange* asserts that every subrange variable contained as a component of its argument is in the correct range.

Definition. Inrange* is a mapping <pascal variable> x <type> → <formula>. For simple types, Inrange*(v, t) is true if Inrange(v, t) is. Inrange*(v, t) is true for a compound type if Inrange*(c, ***type***(c)) is true for every component c of v.

The idea of the Inrange lemma is a characterization of the possible sets of states of programs that always execute without runtime errors. Any actual execution must begin in the outermost block with all variables uninitialized. Data needed by the

program is obtained by a READ procedure which always returns values that are DEF and Inrange. Given that the program always runs without errors, what do we know about the set of all possible states if it terminates? Variables that the program assigns to every time it is run will always be DEF and Inrange* at the end. Variables that are never touched by the program will be completely unspecified at the end. Variables assigned to on some runs but not on others can be ¬DEF at the end, or can have a value dependent on the values of the other variables. If the value is dependent on the other variables, it must be an Inrange* value. The essential point is: If a program determines the value of a variable, the value must be Inrange*. If a variable is always DEF at the end of a program, then it must always be Inrange*.

Definition. Let X be the set of simple components of the declared variables. For instance if v is declared

```
VAR v: ARRAY [1..2] OF RECORD f:INTEGER; g:BOOLEAN END;
```

then X will contain the variables v[1].f, v[2].f, v[1].g, v[2].g. Note that X is a set of variables, not a set of the values the variables. A state of a program is an assignment of values to each of the elements of X. To refer conveniently to the value of a given variable y∈X and the overall state, we will use the notation that the y-form of a state is a pair <z,Z>, where z stands for the value of y, and Z stands for the values of the variables in X-{y}.

A set S of states is DEF-convex for the variable y, iff

for all Z,
$(\forall z\ \langle z,Z\rangle \in S_y \supset DEF(z))$ implies $(\forall w\ \langle w,Z\rangle \in S_y \supset \textit{Inrange}(w, \textit{type}(y)))$.

where S_y is the set of states in S, represented in y-form.

A set of states of X is DEF-convex iff it is DEF-convex for every variable in X. A

formula containing free occurrences of declared variables is DEF-convex iff it is satisfied by a DEF-convex set of states.

Examples: assume the declared variables are
VAR x: INTEGER;
VAR y: 1..10;

(7.1)	True, False	both DEF-convex
(7.2)	y=2	DEF-convex
(7.3)	y=40	not DEF-convex
(7.4)	y≠40	DEF-convex
(7.5)	DEF(y)	not DEF-convex
(7.6)	x=1 ⊃ y=2	DEF-convex
(7.7)	x=1 ⊃ y=40	not DEF-convex

If S is the set of final states of a program that does not have runtime errors, then S is DEF-convex. In the examples, a program can set y to 2, so 7.2 is DEF-convex, but 7.3 cannot be DEF-convex because 40 is out of range. Although y≠40 is DEF-convex, it is not a possible set of final states — the DEF-convex sets include more than final states sets. To attempt to characterize only final states would require much more detail than we need here. Note that 7.5 is too weak to be a final set of states because it includes both 7.2 (a possible set) and 7.3 (an impossible set).

Lemma 7.4a **If a program is started in a DEF-convex set of states and always executes without runtime error, then the final set of states will be DEF-convex.**

It follows that if a program always leaves a variable DEF when it halts, the variable must be Inrange* at the end.

Lemma 7.4b **If B is a Pascal statement, pv is a Pascal variable, P is a DEF-convex predicate, and ⊢ P [[B]] DEF(pv), then ⊢ P [[B]] *Inrange**(pv, ***type***(pv)).**

The restriction on P in this lemma is necessary. Recall that extended semantics does not specify the initial values of variables, and that subrange type variables have the

same sort as the base type of the subrange. Consequently, there is nothing that says a subrange variable cannot be out of range if its value is not assigned by the program. The following formula is a a theorem, even if the variable S declared with a subrange of only 1..100.

$$\vdash S{=}500\ [\![\mathit{empty}]\!]\ DEF(S) \wedge S{=}500.$$

Of course, the extended definition checks that any program that uses the value of S first assigns it a value in the proper range.

Runcheck makes use of a restriction that the entry assertion for the outermost block of a program must be DEF-convex.[6] With this assumption, Runcheck can infer bounds on the value of a subrange variable if it is known to be DEF. In some cases, this can permit lengthly assertions to be omitted. For instance, if a complex data structure contains subrange variables and the entire data structure is DEF, bounds for the subrange variables can be deduced without any additional assertions. By induction on the depth of procedure calls, the lemma can also be applied to formal parameters when reasoning about a procedure body. Since a value parameter v must be DEF on entry, ***Inrange****(v,t) must be true initially. Variable parameters do not have to be DEF on entry, but if the value is used somewhere in a procedure body it must be possible to prove that the variable is DEF and the Inrange lemma applies at that point.

[6] In an actual Pascal program, no assumptions can be made about the initial values of variables declared in an outermost block. To be strictly realistic, the verifier should not permit entry assertions there. They are permitted as a small convenience; a main block with an entry assertion is considered to be a shorthand for a procedure with globals. The significance of this is that the truth of the entry assertion must be assured by some calling program i.e. it is possible to declare a procedure with an entry assertion that is not DEF-convex, but its actual set of entry states is then a DEF-convex restriction of the declared entry condition.

Example 5: Constructing a Spanning Tree.

The following program is a simple algorithm [Se70] for finding a spanning tree of an undirected loop-free graph with E edges and V vertices. If the graph is disconnected, it grows a spanning forest. The graph is entered as a table of edges in the arrays IA and JA, so that the vertices of the k^{th} edge are IA[k] and JA[k]. The program stores the indices of the spanning tree's edges in T[1], . . . ,T[V-P], where P is set to the number of trees in the spanning forest.

This example illustrates the use of subranges and the inrange lemma to strengthen the entry assertion of a procedure. Since IA and JA are tables of vertices, they have been declared as arrays of subrange values 1:V. It is typical in graph manipulating programs to use a value stored in one array to compute an index into another array. Here, the variable I is set to IA[K] and then VA[I] is accessed. For the latter access to be in the subscript range 1:V of VA on every iteration, all elements of IA must have been in the range initially. Because IA and JA are value parameters, their initial values must be DEF, and by the inrange lemma, Runcheck can infer that the elements are in the correct range. Similar reasoning is required for other array accesses.

```
VAR E,V:INTEGER;

PROCEDURE SPANNING(IA,JA: ARRAY[1:E] OF 1:V;
                          VAR P: INTEGER;
                          VAR T: ARRAY[1:V-1] OF INTEGER);
ENTRY DEF(E) ∧ DEF(V) ∧ 1≤E ∧ 2≤V;
EXIT TRUE;
VAR I,J,K,C,N,R: INTEGER;
VAR VA: ARRAY[1:V] OF INTEGER;

BEGIN
C:=0;
N:=0;
FOR K:=1 TO V INVARIANT 1≤K ∧ K≤V+1 ∧ DEFRANGE(1,K-1,VA)
          DO VA[K]:=0;
FOR K:=1 TO E
     INVARIANT 1≤K ∧ K≤E+1 ∧ 0≤N ∧ 0≤C ∧ N≤K-1 ∧ C≤K-1 ∧ K≤V+N-1
  DO BEGIN
     IF K-N=V-1 THEN GOTO 1;
     I:=IA[K];
     J:=JA[K];
     IF VA[I]=0 THEN
     BEGIN
          T[K-N]:=K;
          IF VA[J]=0 THEN BEGIN
                         C:=C+1;
                         VA[J]:=C;
                         VA[I]:=C;
                         END
                         ELSE VA[I]:=VA[J];
     END
     ELSE IF VA[J]=0 THEN
     BEGIN
          T[K-N]:=K; VA[J]:=VA[I];
     END
     ELSE IF VA[I]≠VA[J] THEN
     BEGIN
          T[K-N]:=K; I:=VA[I]; J:=VA[J];
          FOR R:=1 TO V INVARIANT 1≤R ∧ R≤V+1
               DO IF VA[R]=J THEN VA[R]:=I;
     END
     ELSE N:=N+1
  END;
1: P:=V-E+N;
END;
```

Note that IA and JA could have been declared as arrays of INTEGER, and the restriction on the values could have been part of the entry assertion. Expressing the

restriction would involve a quantified assertion such as

$$\forall x\ (1 \le x \le E \supset 1 \le IA[x] \le V).$$

This is both more difficult to write than the subrange type specification, and it causes difficulty in theorem proving.

8. Generalizations of the extended semantics

8.1 Dynamic subranges

There are programming languages more flexible than Pascal, which allow declaration of dynamic subranges. ADA, in particular, has flexible dynamic type declarations. A reasonable extension to Pascal is to permit subrange declarations involving expressions, e.g.

```
TYPE s = 1..2*x;
```

The expressions for the bounds are evaluated each time the scope is entered, and the range of s is fixed for the duration. Dynamic arrays can be obtained by using a dynamic subrange as the index type for an array etc.

The extended semantics can be adopted to handle dynamic subranges by defining Inrange(e, s) to refer to the values obtained when the expressions for the bounds on s are evaluated. The declaration rules for functions and procedures would be changed to check for error free evaluation of the expressions in the type declarations. Also, depending on the restrictions in the programming language, renaming would be needed to distinguish between the initial values of the variables appearing in the type declaration and the values assigned after the dynamic declaration was evaluated.

8.2 Bounds on depth of recursion and dynamic variable allocation

Like the bound for arithmetic overflow, bounds on recursion and heap storage are implementation dependent. In critical applications, the actual bounds may be set in advance, and one might want to verify that the available storage will be sufficient. In

other cases, the particular bound is not important, but it might be useful to verify that a program does not attempt unlimited recursion etc.

To describe bounds on depth of calls, two new undeclared integer variables are introduced in the procedure call rule. The variable Stksize represents the maximum depth of calling; Stkptr represents the current depth. The procedure call rule is modified to enforce a restriction that Stkptr≤Stksize. Neither variable can be assigned to by the program. Stkptr is 0 on entry to a main program, and each level of function or procedure calling increases it by 1. With these additions, the procedure call rule is

for i=1, . . . ,m, P [[Eval Ai]] ***Inrange***(Ai, ti),

for i=1, . . . ,n, P [[Locate Vi]] True,

I(X,Y,G,S) [[Procedure p(X1:t1; . . . ;Xm:tm; VAR Y1:u1; . . . ; VAR Yn:un); B]] O(X,Y,G,S);

P [[Eval A1; . . . ;Eval Am; Locate V1; . . . ;Locate Vn]] ***Disjoint-set***(V ∪ G)
∧ I(A,V,G,Stkptr+1,Stksize)
∧ ∀Z,GB (O(A,Z,GR,GB,Stkptr+1,Stksize)
$$\supset Q\left|\begin{matrix} V1 & \cdots & Vn & GW1 & \cdots & GWk \\ Z1 & \cdots & Zn & GB1 & \cdots & GBk \end{matrix}\right.)$$
∧ Stkptr+1≤Stksize

P [[p(A1, . . . ,Am,V1, . . . ,Vn)]] Q (PC2)

where S stands for the set of variables {Stkptr, Stksize}. Note that in practical applications, it might be important to use some measure of the actual amount of stack space used by a program instead of just the depth of recursion. It would be simple to define a different function that depended e.g., on the number and types of variables in the procedure, for incrementing Stackptr. To measure the heap storage used, counters can be added to the rules for NEW statements.

Example 6: Recursive Tree Traversal.

Type PTR is defined to be a pointer to a record with .A and .B fields of type PTR. The recursive procedure WALK simply does a depth first walk on a tree P. To avoid stack overflow, P must not lead to any cyclic list structure and there must be enough room on the stack for DEPTH(P, #REC) procedure calls, so Stacksize must be greater than or equal to Stackptr+DEPTH(P, #REC). Stackptr and Stacksize are declared as VIRTUAL variables to indicate that they may appear in assertions, but may not be used in executable parts of the program. ACYCLIC and DEPTH are user defined symbols for documenting programs that operate on trees. The assertion DEF(#REC) states that every allocated record in the heap of type REC is fully initialized. This assures that WALK will not encounter uninitialized dynamic variables.

```
TYPE PTR=↑REC;
   REC=RECORD A:PTR; B:PTR END;

VIRTUAL VAR Stackptr, Stacksize: INTEGER;

PROCEDURE WALK(P:PTR);
ENTRY ACYCLIC(P, #REC) ∧ DEF(#REC) ∧ Stacksize ≥ Stackptr+DEPTH(P, #REC);
EXIT TRUE;

BEGIN
IF P≠NIL THEN BEGIN WALK(P↑.A); WALK(P↑.B) END;
END;
```

The proof depends on two lemmas about acyclic list structure. If p is a pointer to acyclic list structure in the reference class ↑r, then p↑.f points to acyclic list structure. If p points to acyclic list structure, then the depth of p↑.f is less than the depth of p.

```
ACYCLIC(p, #r) ∧ p≠NIL  ⊃  ACYCLIC(p↑.f, #r)
ACYCLIC(p, #r) ∧ p≠NIL  ⊃  DEPTH(p↑.f, #r) ≤ DEPTH(p, #r)-1
     (where .f is .A or .B)
```

The lemmas are provided by the user to the system in the form of inference rules [SVG79] to be used by the theorem prover.

8.3 Procedure Valued Parameters

Procedure (and function) valued formal parameters in Pascal have the weakness that the arguments of formal procedures are not declared. It is not possible to determine syntactically whether a procedure valued formal parameter is called with the right number and type of arguments. It is a simple matter to tighten the language by introducing more detailed declarations; if this is done, the usual syntactic checks can be performed for procedure valued parameters, and they can be included in the axiomatic definition.[7] As an example of a program using more detailed declarations, Sum(a,b,f) computes the sum of f(x) when x ranges from a to b.

```
FUNCTION Sum(a,b:INTEGER; f:FUNCTION(INTEGER):INTEGER): INTEGER;
VAR i,s:INTEGER;
BEGIN
  s:=0;
  FOR i:=a TO b DO s:=s+f(i);
  Sum:=s
END;
```

Clarke [Cl79] shows that any sound and complete axiomatic definition of procedure valued parameters in a language with recursion, static scoping, read write global variables, and internal procedure declarations, must depend on some method of making assertions about the state of the runtime stack of local variables. Such an approach would greatly complicate both the semantic definition and the process of specifying and verifying programs. Instead, we will make the restriction that functions or procedures with globals may not be passed as parameters. With this restriction, procedure valued parameters can be introduced in a natural manner.

[7] This section discusses extensions planned but not yet implemented in the verifier. A treatment of the consistency and completeness of our axiom system for procedure valued parameters without global variables is in preparation.

The specification method will be to declare an Entry and Exit assertion for each formal parameter; these will be used in the ordinary call rules when the formal is called. When a procedure parameter is passed, the call rules will check that the actual satisfies the declared specifications of the formal.

Nesting of procedure parameters is permitted to any finite depth. Thus a procedure can have a procedure parameter which takes another procedure as one of its parameters, but self application of procedures is not possible. The various possibilities are illustrated in the example below: a procedure p has value parameters U, variable parameters V, a function parameter s, and a procedure parameter q. The procedure q takes a function parameter r.

The main specification given for p is a set of entry-exit assertions, Ip and Op. An occurrence in the assertions of the formal function parameter s as a function sign stands for the value of the functional parameter, and not for a constant function. The assertions may be thought of as first order schemes, which the procedure call rule adopts to particular calls by substituting the actual function sign for the formal s. To distinguish this kind of substitution from sustitution for free variables, the following notation will be used.

Notation: Q[f](X) is a formula containing the function sign f and free variables X. After a particular formula Q[f](X) has been introduced, we will write Q[g](Y) to stand for the result of replacing the function sign f by g and substituting Y for X in Q.

Each formal procedure parameter has a declaration in p of its entry-exit assertions. The declarations are like ordinary procedure declarations, except that the reserved word FORMAL is used in place of the procedure body. Since the formal parameter q takes a function r as an argument, the declaration of q has a declaration for r nested inside it.

Declarations with procedure and function formals.

```
PROCEDURE p(U; VAR V;
        FUNCTION s(Y):t;
        PROCEDURE q(W; Function r(Y):t));

    FUNCTION s(Y):t;        % specifications of formal parameter s %
    ENTRY Is(Y);
    EXIT Os[s](Y,s);
    FORMAL;

    PROCEDURE q(W; Function r(Y):t);        % specifications of q %
        Function r(Y):t;  % specifications of formal parameter of q %
        ENTRY Ir(Y);
        EXIT Or[r](Y,r);
        FORMAL;
    ENTRY Iq[r](W);
    EXIT Oq[r](W);
    FORMAL;

GLOBAL GR, VAR GW;
ENTRY Ip[s](U,V,G);
EXIT Op[s](U,V,G);          % specifications of p %

BEGIN pbody END;            % executable statements of p %
```

In this example, the Entry and Exit specifications for p state that the value parameters U, variable parameters V, function parameter s, and global parameters G, must satisfy Ip[s](U,V,G) on entry to p, and Op[s](U,V,G) on exit. Furthermore, the actual parameter supplied for s must have the property that if Is(Y) holds for the value parameters Y to s, then Os will hold for the result of S. The specifications for q are similar, but have further specifications for r nested within them in the same way that the specifications for s are nested in p.

Notation: In the following rules, entry-exit assertions enclosed in brackets, ≪I,O≫, are included in the procedure headers as an abbreviation for the full procedure declarations as shown above.

The idea of the declaration rule is to use the declared entry exit specifications of the formal parameters, in this case s and q, to prove the specifications for p. Then for

calls to p, the call rule will check that the actual function and procedure parameters satisfy the specifications declared for s and q.

The following example of the declaration rule states that we can infer that Ip and Op are valid entry exit specifications for p if it is possible to prove that Ip and Op are valid for the body of p (8.3), under the assumptions (8.1 and 8.2) that s has specifications ⟪Is,Os⟫ and q has specifications ⟪Iq,Oq⟫.

Example Procedure declaration.

$$\{Is(Y)\ [\![\text{Function } s(Y):t;\ \text{FORMAL}]\!]\ Os[s](Y,s), \quad (8.1)$$
$$Iq[r](W)\ [\![\text{Procedure } q(W;\ r:\langle\!\langle Ir,Or\rangle\!\rangle);\ \text{FORMAL}]\!]\ Oq[r](W)\} \quad (8.2)$$
$$\vdash Ip[s](U,V,G) \wedge DEF(U) \wedge \textit{Inrange}(UI,tI)\ [\![pbody]\!]\ Op[s](U,V,G) \quad (8.3)$$

$$Ip[s](U,V,G)$$
$$[\![\text{Procedure } p(U;\ V;\ s:\langle\!\langle Is,Os\rangle\!\rangle;\ q(W;\ r:\langle\!\langle Ir,Or\rangle\!\rangle):\langle\!\langle Iq,Oq\rangle\!\rangle);\ pbody]\!]\ Op[s](U,V,G)$$

If s and q were actual defined subprograms (instead of formals), any properties of them needed for proving p could be deduced from their definitions by the declaration rule. But the actual bodies corresponding to s and q are not fixed. The declaration rule for p compensates for this by allowing us to introduce assumptions about s and q into the proof of p. These assumptions must then be justified for the actual parameters whenever p is called; this is done in the call rule.

Example Procedure call.

$$\text{for } i=1,\ldots,m,\ P\ [\![\text{Eval Ai}]\!]\ \textit{Inrange}(Ai, ti), \tag{8.4}$$

$$\text{for } i=1,\ldots,n,\ P\ [\![\text{Locate Bi}]\!]\ \text{True}, \tag{8.5}$$

$$Ip[s](U,V,G)\ [\![\text{Procedure p(U; V; s:}\langle\text{Is,Os}\rangle\text{; q(W; r:}\langle\text{Ir,Or}\rangle\text{):}\langle\text{Iq,Oq}\rangle\text{); pbody}]\!]\ Op[s](U,V,G), \tag{8.6}$$

$$Is(Y)\ [\![\text{Function c(Y):t; cbody(Y)}]\!]\ Os[c](Y,c), \tag{8.7}$$

$$Iq[r](X)\ [\![\text{Procedure d(X; r:}\langle\text{Ir,Or}\rangle\text{); dbody(X,r)}]\!]\ Oq[r](X), \tag{8.8}$$

$$\frac{P\ [\![\text{Eval A1; }\ldots\text{;Eval Am; Locate B1; }\ldots\text{;Locate Bn}]\!]\ \textit{Disjoint-set}(B \cup G) \wedge Ip[c](A,B,G) \wedge \forall Z,GB\ (Op[c](A,Z,GR,GB) \supset Q\Big|^{B1\ \ldots\ Bn\ \ GW1\ \ldots\ GWk}_{Z1\ \ldots\ Zn\ \ GB1\ \ldots\ GBk})}{P\ [\![p(A,B,c,d)]\!]\ Q} \tag{8.9}$$

For the procedure call, conditions 8.4, 8.5 and 8.6 are as before. Condition 8.7 checks that the actual function parameter c satisfies the specifications of s; 8.8 checks the entry-exit assertions for the actual procedure d. In 8.7, cbody(Y) stands for the body of the actual parameter c, if c is a declared function in the context of the call. In case c happens to be a formal parameter of another procedure, say q, cbody(Y) is taken to be the reserved word FORMAL, and 8.7 can be justified by the assumption about c in the declaration rule for q.

Crucial to these two rules are the type declarations, which syntactically enforce the requirement that each subprogram parameter accept only a fixed type and hence only a fixed depth of nesting of formal parameters. In the example, s has no procedural parameters; let us call this depth zero. Then the depth of q is one, and of p two. Because of the type declarations, each actual parameter to a subprogram must have the same depth as the corresponding formal. Observe that this prevents self application of procedures, which could lead to circular proofs such as would occur if an assumption about p was used to deduce a property of p in the declaration rule.

The rules are justified by the fact that each assumption introduced for a formal parameter in the declaration rule is verified for the corresponding actual in the call rule. Note that in any execution, the actual value of each formal parameter must be traceable back to a declared (non formal) subprogram with the same depth.

It can be easily seen that the two new rules are only a means of transferring and rebinding entry exit specifications which must eventually be justified using the original rules without procedure parameters. Consider the case of a procedure p which has a formal parameter s declared as FUNCTION s(Y):t, so that p's depth of nesting of formals is one. The actual value supplied for s may be passed to p through many levels of procedure calls, but ultimately any specifications for s must be proven with the ordinary declaration rule. Thus any specifications that can be proven for p are ultimately based on the ordinary function declaration rule. Similarly, the specifications of a procedure q of depth two are based on specifications of procedures of depth less than two. In this way, all deductions with the two new rules can be traced back to the ordinary rules. What has been added is the ability to transfer specifications, corresponding to the added capability in the language for transferring declared procedures by parameter passing.

9. Discussion

Our definition of Pascal describes some important aspects of the language that have not been included in previous axiomatic definitions. We began by recalling that a proof of P {A} Q does not give any assurance that a program will be free from runtime errors, and argued that a stronger relation, P [[A]] Q, is a better indicator of program reliability. As part of our presentation of Pascal semantics, we have developed a precise and comprehensive definition of the evaluation of expressions and Pascal variables, using partial correctness statements to account for function calls within expressions. Previous axiomatic definitions have not dealt fully with the semantics of function calls within expressions. We then used the definition of evaluation to define Pascal statements, procedures and functions. The complete definition is very concise, although it captures many complicated details of the language. One of the crucial advantages of our axiomatic technique is its simplicity; absent are the clouds of obscuring notation commonly found in denotational definitions. The clarity and simplicity of our approach are of greatest importance when the definition is actually used to verify programs; because program specifications and the proofs are also simple and understandable, the user is free to concentrate on the real issues surrounding a program and its correctness.

Our axiomatic definition has been part of a development with the goal of building a useful automatic verifier. This has influenced the definition in several ways. One important requirement for useful verification is to have convenient methods for specifying programs. In Runcheck, specifications are greatly simplified by having a single predicate, DEF, as the basis of all predicates referring to variable initialization. The Lessdef and Inrange lemmas also eliminate the need for certain kinds of detail in specifications. Although the idea of derived inference rules is by no means new, this

technique is more useful in practice than has been previously realized.

Appendix 1-A: Development of the WHILE Rule

This section explains the actual While rule used in Runcheck. The rule of section section 6.2,

```
P ⊃ I,
I [[Eval B; ASSUME B; S]] I,
I [[Eval B]] ¬B ⊃ Q
------------------------------                    (WHILE1)
P [[INVARIANT I WHILE B DO S]] Q
```

does not help to reduce the need for detailed invariants and is not convenient to use in practice. The implemented rule has four additional features:

1) It adds an invariant referring to the evaluation of the While test, B. B is evaluated once on each iteration, and so it must be an invariant of the loop that B can evaluate safely.

2) It makes it unnecessary for the invariant to refer to variables which cannot be changed in the loop. This has been previously called a *frame axiom* [ILL75, Su76].

3) It applies the Lessder lemma, adding to the invariant the information that variables changed on the loop cannot become less fully initialized.

4) Runcheck's automatic documenter generates invariants which are valid in the unextended semantics. Because proofs in the extended semantics can be separated, with part done in the ordinary semantics (Specification lemma), the extended While rule can assume the validity of documenter invariants without reproving them.

We now discuss the implementation of these changes.

1) From the definition of P [[Eval e]] Q, one can write down a sufficient precondition

for e to evaluate without error. This formula will be called PRE[Eval e; True]. As an example, if the test of a While loop is f(a)+b≤0 and f has the declaration

```
FUNCTION f(x: INTEGER): c:d;
ENTRY I(x);
EXIT  O(x);
...
```

then the condition

```
PRE[Eval f(a)+b≤0; True]
  ▪ DEF(a) ∧ DEF(b) ∧ I(a)
              ∧ (O(a) ∧ DEF(f(a)) ∧ c≤f(a)≤d  ⊃  -MAXINT≤f(a)+b≤MAXINT)
```

is added as an invariant of the loop.

2) The variable identifiers are divided into a subset X which are not changed in the body of the loop and a subset Y which may be changed. A set of new unique variables, Y', is introduced. The extended form of the frame rule is

```
P(X,Y) ⊃ I(X,Y),
P(X,Y)∧I(X,Y') [[Eval B(X,Y'); Assume B(X,Y'); S(X,Y')]] I(X,Y'),
P(X,Y)∧I(X,Y') [[Eval B(X,Y')]] ¬B(X,Y') ⊃ Q(X,Y')
-------------------------------------------------------------------------
P(X,Y) [[Invariant I(X,Y) While B(X,Y) Do S(X,Y)]] Q(X,Y)
```

where the Y variables stand for the values of variables before the loop and the Y' variables stand for the values of variables during or after the loop.

3) For each variable, y, which can be changed in the body, Lessdef(y, y') can be assumed to be a valid invariant.

4) Documenter invariants D(X,Y,Y') can be assumed valid.

The final rule is:

$P(X,Y) \supset I(X,Y) \wedge PRE,$

$P(X,Y) \wedge I(X,Y') \wedge PRE \wedge Lessdef(Y,Y')$
$\quad \wedge D(X,Y,Y') \; [\![Eval\ B(X,Y');\ Assume\ B(X,Y');\ S(X,Y')]\!]\ I(X,Y') \wedge PRE,$

$P(X,Y) \wedge I(X,Y') \wedge PRE \wedge Lessdef(Y,Y')$
$\quad \wedge D(X,Y,Y') \; [\![Eval\ B(X,Y')]\!]\ \neg B(X,Y') \supset Q(X,Y')$

$P(X,Y)\ [\![Invariant\ I(X,Y)\ While\ B(X,Y)\ Do\ S(X,Y)]\!]\ Q(X,Y)$ (WHILE2)

where PRE is PRE[Eval B;TRUE].

Appendix 1-B: Simultaneous Substitution for Disjoint Variables

In this section, we present the definitions of disjointness for Pascal variables and simultaneous substitution for disjoint Pascal variables. To begin, we need to define the translation of a Pascal variable into a standard representation as a sequence consisting of a main variable identifier followed by zero or more selectors. In the following, <e1, . . . ,en> denotes a sequence of n terms, and the operator • stands for concatenation of finite sequences.

The function Seq(v): <Pascal variable> → <term sequence> is defined as follows:

Seq(id) = <id> if id is an identifier
Seq(v.f) = Seq(v) • <.f>
Seq(v[i]) = Seq(v) • <i>
Seq(v↑) = <#t, v> where #t is the reference class

Definition of Disjoint(v, w)

Let v and w be Pascal variables and Seq(v) = <v0, . . . ,vn>, Seq(w) = <w0, . . . ,wm>, and assume m≤n. Then Disjoint(v, w) is the following formula:

if v0 and w0 are distinct identifiers, then Disjoint(v, w) → True;

otherwise, Disjoint(v, w) → (v1≠w1 ∨ . . . ∨ vm≠wm)

The current implementation of Runcheck uses a much more restrictive definition of disjointness (it only compares v0 and w0); this restriction is not essential and will be removed in a later version.

Simultaneous Substitution

We can now define a simultaneous substitution of n terms e1, . . . ,en for disjoint

v1, . . . ,vn. Let Seq(vi) = <vi_0, . . . ,vi_{mi}> for i = 1, . . . ,n. Let t1, . . . ,tn and di_j for i = 1, . . . ,n, j = 1, . . . ,mi, be new identifiers not appearing in P, the vi or the ei.

Define Unseq: <term sequence> → <Pascal variable> to be the inverse of Seq; Unseq(Seq(v)) = v.

Then we can define

$$P\Big|_{e1}^{v1} \cdots {}_{en}^{vn}$$

$$= P\Big|_{t1}^{\textit{unseq}(<v1_0,d1_1,\ldots,d1_{m1}>)} \cdots \Big|_{tn}^{\textit{unseq}(<vn_0,dn_1,\ldots,dn_{mn}>)}$$

$$\cdots \Big|_{e1}^{t1} \cdots \Big|_{en}^{tn} \quad \cdots \quad \Big|_{v1_1}^{d1_1} \cdots \Big|_{v1_{m1}}^{d1_{m1}} \quad \cdots \quad \Big|_{vn_1}^{dn_1} \cdots \Big|_{vn_{mn}}^{dn_{mn}}$$

Example B.1: Simultaneously swapping a[i] with a[j] and changing i.

$$P(a,i,j)\Big|_{a[j]\ a[i]\ i+1}^{a[i]\ a[j]\ i}$$

$$= P(a,i,j)\Big|_{t1}^{a[d1]} \Big|_{t2}^{a[d2]} \Big|_{t3}^{i} \Big|_{a[j]}^{t1} \Big|_{a[i]}^{t2} \Big|_{i+1}^{t3} \Big|_{i}^{d1} \Big|_{j}^{d2}$$

$$= P(<<a, [j], a[i]>, [i], a[j]>, i+1, j)$$

Note that <<a, [j], a[i]>, [i], a[j]> stands for the value of the array a after first assigning the value a[i] to the jth position, and assigning a[j] to the ith position.

Example B.2: Swapping two variables accessed by pointers.

Consider the effect of simultaneously interchanging x↑ and y↑, where x and y are pointer variables.

```
TYPE ptr = ↑INTEGER;
VAR x, y: PTR;
```

$$P(x, y, \#INTEGER)\Big|\begin{matrix}\#INTEGER\langle x\rangle & \#INTEGER\langle y\rangle \\ \#INTEGER\langle y\rangle & \#INTEGER\langle x\rangle\end{matrix}$$

$$= P(x, y, \langle\langle \#INTEGER, \langle y\rangle, \#INTEGER\langle x\rangle\rangle, \langle x\rangle, \#INTEGER\langle y\rangle\rangle)$$

The final value of the reference class #INTEGER is exactly analogous to the final value of the array a in example B.1.

Chapter 2. Verification with Variant Records, Unions, and Data Representation Mappings.

The challenge in programming language design comes from the interplay between conflicting concerns of generality, efficiency, reliability and elegance. In this chapter, we apply the idea of the error checking axiomatic semantics to Pascal variant records. The main rationale for providing variant records was to enable programs to use less space than would be used with ordinary records. It is well known that there is an apparent flaw in the design of variants in Pascal: they can be used as a loophole to violate the type restraints of the language. In most situations, enforcement of typing contributes to reliability by preventing simple programming errors. We will see, though, that a loophole in typing can be used in ways which contribute to the efficiency and generality of the language.

Section 1 of this chapter introduces Pascal variants and their applications. Variants can be added to our error checking semantics if we prohibit type violations. We will define a new error, variant access error, which occurs when the value of a variant record is used in a way which would violate typing. It will then be possible to prove in the extended semantics that no type violations occur.

This would be the end of the story if reliability was the only concern; however, we will see that there are also implementation problems related to variants and we would like to preserve the benefits of intentional use of the loophole.

Thus we propose to replace variants by two new language features. Union data types, to be discussed in section 2, permit a variable to have different formats at different times without permitting type violations and without the other implementation problems of variants. In section 3 we will consider a separate mechanism for intentional conversion between values of different types. Interconversion between two

different types will be permitted under controlled conditions to prevent the construction of invalid values. The combination of these two new mechanisms retains the generality of variants while offering a higher level of reliability.

1. Variant Records

Although the discussion refers to variants as they appear in Pascal, all of our remarks will apply to other languages with a similar notion of data type. Types in Pascal are quite conventional; there are a number of primitive types (e.g. BOOLEAN, INTEGER, CHARACTER, REAL), and then defined types:

```
Enumerated - E = (c1, . . . , cn);

Subrange - S = c1 . . c2;

Array - A = ARRAY [s] OF t;

Record - R = RECORD id1:t1; . . . ; idn:tn END

Pointer - P = ↑t;
```

Another important assumption in the discussion is that variable declarations are *strongly typed*. This will be understood to mean that the range of values of a variable is restricted to permit efficient compilation. In Pascal, for instance, strong typing means that every variable or expression has a single Pascal type which can be determined statically from the type and variable declarations. A compiler takes advantage of strong typing by generating code that is efficient for the expected range of values, and which may not even have the correct function outside of the range.

In Pascal, variant record definitions have the form:

```
V =   RECORD
          f:t;
          .
          .
          .
          f:t;
          CASE tagid:e OF
               c1: (f:t;  . . . f:t);
               .
               .
               .
               cn: (f:t;  . . . f:t);
     END;
```

where e is an enumerated type or subrange, and the ci are constants of type e. The CASE clause is called the *variant part*. The variable tagid is of type e, and is optional.

A variant record provides a single type having several different formats. Each case in the variant part is a possible format. All the fields preceding the variant part are always present. In the variant part, one of the cases can be selected at any time, and only the fields for that case are present.

The various cases are represented in storage as overlapping variables. Thus when the fields for one case are used, the fields for the other cases may get overwritten with meaningless data.

For example, compare the type R, an ordinary record type with three components, with V, a variant record type:

```
R = RECORD A:t1; B:t2; C;t3 END;

V = RECORD A:t1; CASE BOOLEAN OF TRUE:(B:t2); FALSE:(C:t3) END;
```

The variant record always has an .A field, and depending on which case is current, has either a .B field or a .C field. In this example, there is no tag field. It is not possible to tell from the variable itself which case is being represented. Even if a tag field is used, Pascal does not guarantee that the tag will have the correct value. It is

up to the user to set the tag, and there is nothing to prevent access to one of the non-current fields.

1.1 Uses of Variants

The most common use of variants is to allow uniform access to records with different structures. Because of strong typing, it is not ordinarily possible for one variable to range over records with different structures. Variants provide a single type that satisfies the requirements of strong typing. In the previous example, type V includes records with either .A and .B or .A and .C. This is useful in data processing applications, for instance, to create a file of records in which different details are stored depending on the individual. An ordinary record with three fields can always be used in place of the variant in this application, but it would take more space.

There are other important uses of variants, but they are less respectable. For various reasons, one sometimes wants to violate typing by taking a value of one type and interpreting it as another type. In Pascal, variants are a loophole for such violations, because it is possible to select a field from the wrong case.

As an example, consider how variants can be used to convert between pointers and integer values.

```
TYPE ptr = ↑t;
TYPE v = RECORD CASE tag:BOOLEAN OF
            TRUE:(f:ptr);
            FALSE:(g:INTEGER)
            END;

VAR p:ptr;  x:v;  n:INTEGER;

BEGIN
      NEW(p);
      x.tag:=TRUE;
      x.f:=p;
      x.tag:=FALSE;
```

```
    n:=x.g
END;
```

Since Pascal does not define input or output operations for pointer values, a user with knowledge of the language implementation might use variants to convert between ptr and integer values. This fragment might be used under the assumption that variables of types ptr and integer occupy the same amount of space. A pointer variable p is initialized by a NEW statement, and then because x.f and x.g overlap in storage, the pointer value can be stored in the integer variable n.

In general, a *variant access error* will be said to occur when accessing the value of a field which has been changed by an assignment to an overlapping variable in another case. The access error can illegally "convert" between any two types.

Obviously, conversions of this kind are very dangerous and to use them without sufficient precaution is poor programming practice. Pascal can be criticized for permitting insecure conversions. In the next section, we introduce a union construct that does not have this problem. On the other hand, occasionally there is a legitimate need for conversions that are not defined in Pascal. One could argue that Pascal's success as a systems programming language is in part due to its flexibility — permitting the type violations in a few critical places.

The type violations are needed infrequently, but when they are needed they can be a major factor in the efficiency or generality of a program. For instance, on machines without floating arithmetic hardware, certain operations on reals can be done more efficiently by special purpose bit operations than by the general floating point routines. The bits of real variable can be directly accessed in most Pascal implementations by illegally converting to type SET 1 . . n OF BOOLEAN, where n is the word size. This trick depends on the fact that sets are represented as packed bit vectors.

Insecure conversions for the sake of generality are sometimes needed in systems programming. In an operating system written in a high level language, such as Brinch Hansen's SOLO written in Concurrent Pascal [BH77], there may be low level operations on storage that are applicable to all types. A procedure for transferring a page to a disk can use any block of the right size, regardless of the type of the variables stored there. Concurrent Pascal has a special provision for this kind of conversion: a formal parameter can be declared UNIV, meaning the actual must match in internal size, but not in type.

This line of thought suggests that unions alone are not a complete replacement for variants. Naturally, permitting insecure conversions raises a number of language design issues. How should the meaning of the conversion be defined? What restrictions are needed, and how can they be enforced? An approach to these issues will be discussed in section 3, where we introduce further language extensions for uniform access to arbitrary types. These operations sometimes have complex preconditions that are expensive to test at runtime. Since they are used in few places in a program, it is reasonable to verify correct useage.

Much of verification's impact on language design has been to suggest restrictions that make verification more practical. But verification can also lead to the removal of restrictions: the programmer can be given certain kinds of freedom that are not usually present in high level languages, with a verifier to check that the new operations are used safely.

1.2 Assignment and selection on variant records

This section presents the axiomatic definition of the assignment and selection operations for ordinary records and then considers the differences with variant records. Variant access error is defined. Some of the properties of standard records

do not hold for variants. The variant selection error results in an undefinable state, making it necessary to restrict program executions.

The basic operations associated with record variables are selection and assignment of components. The value of a record variable is determined by the values of its components. This was expressed by the axiom EQa:

EQa) $x=y \equiv (x.f1=y.f1 \wedge \ldots \wedge x.fn=y.fn)$
where f1, . . . fn are the field names for record type t.

The notation <r, .f, e> stands for the record r after assigning r.f:=e. In this notation, assignment to a component is defined by:

$$P(\langle a, .f, e\rangle)\ \{a.f := e\}\ P(a).$$

For non-variant records, the assignment operator has the following property:

REC1) $\langle a, .f, e\rangle.g = \text{IF } .f=.g \text{ THEN } e \text{ ELSE } a.g$

and the following familiar properties which are consequences of REC1 and the definition of equality:

REC2) $\langle a, .f, a.f\rangle = a$

REC3) $\langle\langle a, .f, e\rangle, .f, g\rangle = \langle a, .f, g\rangle$

REC4) $.f\neq.g \supset \langle\langle a, .f, e\rangle, .g, h\rangle = \langle\langle a, .g, h\rangle, .f, e\rangle$

(In writing a field selector .f, f is understood to be a variable ranging over identifiers, and .f=.g if f and g are the same identifier.)

We will now try to adopt the record axioms to variants by making restrictions which leave undefined certain operations on overlapping fields. For convenience, assume that we are considering a variant type having ordinary fields f1, . . . ,fn, and that each variant case has only one field from among c1, . . . ,cn. To begin, we must define selection and equality on a variant field.

VREC1) ⟨a, .c, e⟩.c = e

⟨a, .f, e⟩.c = a.c

⟨a, .c, e⟩.f = a.f

The first line leaves undefined the result of selecting a variant field other than the one which has been most recently been assigned. The second and third parts state that ordinary fields are disjoint from the variant fields.

We could either define equality in the same way as for ordinary records, or say that two variant records are equal if all of the ordinary fields are equal and the same variant was last assigned in both records and to the same value. Note that Pascal's equality operator does not apply to compound types, so it is irrelevant that the second definition would be expensive if implemented. In fact, the definition of equality used for ordinary records would not be very useful for variants because with the definition of variant selection, there is no way to reason about the value of a variant field after another variant field has been assigned to. The result is that equality would not be provable in most cases.

Consequences REC2 and REC3 continue to apply without change, but REC4 does not apply to variants. It states that the order of assignments to different fields does not affect the final value of a record, which is not true if fields overlap.

Thus far we have a first order theory of variants corresponding to the theory of ordinary records without error checking. We can now generalize the error checking semantics to include variants if we can do three things: define what it means for a variant record to be DEF, and give inference rules for Eval v.c and Locate v.c. We have previously defined the semantics of Pascal statements in terms of Eval and Locate, so that once we have the proper definitions of DEF, Eval, and Locate, the semantics will generalize to programs with variants. Recall that a variant access error occurs when a program attempts to use the value of the wrong field; we can prove absence of the errors by giving a sufficiently restricted definition of Eval v.c.

A variant record will be DEF if one of its variant fields is DEF.

DEF3d) DEF(v) ▪ DEF(v.f1) ∧ . . . ∧ DEF(v.fn) ∧ (DEF(v.c1) ∨ . . . ∨ DEF(v.cm))

With this definition, we will be able to use inference rules E1 and L2 without change for variants. They are repeated below in the case of v.c:

$$\frac{P\ [\![\text{Locate v.c}]\!]\ DEF(v.c) \wedge Q}{P\ [\![\text{Eval v.c}]\!]\ Q} \qquad (E1)$$

$$\frac{P\ [\![\text{Locate v}]\!]\ Q}{P\ [\![\text{Locate v.c}]\!]\ Q} \qquad (L2)$$

Observe that variant access error is prohibited because there is no way to show DEF(<a,.ci,e>.cj) if ci is different from cj. In conclusion, the concept of DEF is sufficient to guarantee safe accessing of variant records.

1.3 Practical problems with variants.

The inclusion of variants in Pascal is a design flaw that makes it impossible to implement garbage collection for dynamic variables. In a type such as

```
Type v=RECORD CASE BOOLEAN OF
       TRUE:(f:INTEGER);
       FALSE:(g:↑t)
     END;
```

it is not possible to determine at runtime whether to trace the .g field of a variable during garbage collection marking. Another factor that prevents garbage collection is that pointer variables do not have an initial value.

A special feature of Pascal's NEW statement permits the case of a dynamic variant variable to be permanently set when the variable is allocated. The minimum amount

of storage needed for the particular case can be allocated. This is less than the space for the variant record, which is the maximum space for all cases. The restrictions needed to prevent disasterous errors involving this feature are difficult to enforce at runtime.

```
TYPE rec = RECORD CASE tag: e OF a: ( . . . ); b: ( . . . ) END;
   ptr = ↑ rec;

VAR p, q: ptr;

NEW(q);        allocate variable of type rec

NEW(p, a);     allocate variable with variant fixed to a.
```

Since p↑ may be a variable which occupies less space than q↑, assignments to p↑ must be executed carefully or adjacent variables will be overwritten. In particular, assignments p↑:=v should be permitted only if v is case a, and p↑.b:=v should not be permitted. Note that if p↑ is passed as a VAR parameter, the restrictions must be observed inside the called procedure. To implement this, it would be necessary to associate extra information with all variant variables so that the restrictions could be detected at runtime.

In principle, it is possible to treat these restrictions as runtime errors and verify their absence. To do so, it is necessary to change the model of data structures. The restrictions are a function of the location of a variable, not its value as percieved at the user level. The increased complexity that would be needed in the model would not be justified by this feature alone, although formalization of locations in the underlying logic would have other benefits such as a practical basis for verifying programs with aliasing.

2. Unions.

This section introduces the Union data type. The combination of unions and necessary restrictions on aliasing gives a language in which access errors can be readily detected at runtime, and without the other practical problems associated with variant records.

A UNION type declaration has the form

```
TYPE untype = UNION a1: t1; . . . ; an: tn END;
```

where the ti are types and the ai are constants of an enumerated type or integer subrange. If the ai are of an enumerated type, the type must have been declared previously, and each of its elements must appear once in the UNION declaration.

Assuming that u and u1 are variables of a union type untype above and x is a variable of one of the ti types, then the following operations are defined:

```
VAR u, u1: untype;
    x: t1;
```

SELECTION u:ai returns the ai component of u.

At any time, only one of the components of u exits. Selection of u:ai is an error if the tag of u is not ai. The error can be detected at runtime because the tag always has the correct value.

TAG function TAG(u) returns one of the constants ai, the current tag.

CONSTRUCTORS untype:ai(x) returns a value of untype with tag ai.

As a consequence of the declaration of untype, separate constructor functions are defined for each of the ai. The constructor untype:ai takes values of type ti and converts them into values of the union type.

```
ASSIGNMENTS

    u := u1;

    u:ai := x;        valid only if TAG(u)=ai

    u := untype:ai(x);

    u := x;           implicitly applies construction
```

Assignment to a union variable of a value of the same type is always permitted. An assignment to a component of a union variable, as in the second statement, is permitted only if that component currently exists in u. In the third statement, u is set to the union value constructed from the value of x. The fourth statement is equivalent to the third one: it is possible to determine from the mismatch between the types of u and x, that the constructor untype:ai must applied.

Example: The data structure and basic operations of LISP as defined in Pascal with union types.

```
TYPE TAGS = (A,D,N);
   LISP = ↑U;
   DTPR = RECORD
             CAR: LISP;
             CDR: LISP
          END;
   ATOM = RECORD
             VALUE: LISP;
             PLIST: LISP
          END;
   U    = UNION
             D: DTPR;
             A: ATOM;
             N: INTEGER
          END;

PROCEDURE CONS(X,Y: LISP; VAR RESULT: LISP);
GLOBAL (VAR U);
EXIT TAG(RESULT↑)=D ∧ RESULT↑:D.CAR=X ∧ RESULT↑:D.CDR=Y;
VAR CELL: DTPR;
BEGIN
```

```
  NEW(RESULT);
  CELL.CAR:=X;
  CELL.CDR:=Y;
  RESULT↑:=U:D(CELL)
END;

FUNCTION CAR(X: LISP): LISP;
GLOBAL ( U);
ENTRY TAG(X↑)=D;
EXIT TRUE;
BEGIN
  CAR:=X↑:D.CAR
END;

PROCEDURE PLUS(X,Y: LISP; VAR RESULT: LISP);
GLOBAL(VAR  U);
ENTRY TAG(X↑)=N ∧ TAG(Y↑)=N;
EXIT TAG(RESULT↑)=N ∧ RESULT↑:N=X↑:N+Y↑:N;
BEGIN
  NEW(RESULT);
  RESULT↑:=X↑:N+Y↑:N;
            % note implicit application of U:N() to
              convert INTEGER to type U %
END;
```

2.1 Aliasing Restriction for Unions

If aliasing is permitted, it is possible to subvert runtime tag checking in the language implementation by binding one case of a union variable and then changing the case with a global assignment.

```
TYPE intorchar: UNION 1:INTEGER; 2:CHAR END;
VAR u: intorchar;

PROCEDURE p(VAR x:INTEGER);
GLOBAL(VAR u)
VAR c:CHAR;
 . . .    u := u:2(c)    % changes global value of tag to 2 % . . .
                         % but, note x is still bound to u:1 %


BEGIN          % in main procedure %
 . . .
u := 512;      % sets u to INTEGER case, TAG(u) = 1 %
p(u:1);
 . . .
END
```

This example achieves an illegal overlap between types INTEGER and CHAR, because after the assignment in procedure p, the integer parameter x will overlap with the CHAR case of intorchar.

2.2 Axiomatic definition of Unions.

The value of a union variable u is a function of the tag and the current component.

U1) TAG(u) = t ⊃ u = untype:t(u:t)

Constructors and the tag function have the additional properties:

U2) (untype:t(x)):t = x

U3) TAG(untype:t(x)) = t

Assignment to a union component u:t is defined only if the tag of u is already equal to t before the assignment. The tag remains unchanged after an assignment to u:t. To change the tag, it is necessary to replace the entire union variable.

U4) TAG(u) = t ⊃ ⟨u,:t,e⟩:t = untype:t(e)

U5) TAG(⟨u,:t,e⟩) = TAG(u).

Some consequences of the definition of assignment are:

⟨u,:t,e⟩:t = e

⟨u,:t,u:t⟩ = u

⟨⟨u,:t,e⟩,:t,f⟩ = ⟨u,:t,f⟩

The restrictions on unions in programs are expressed as for variants, by defining DEF, Eval, and Locate.

DEF3e) DEF(x) ⊃ DEF(untype:t(x))

$$\frac{P\ [\![\text{Locate } u{:}t]\!]\ \text{DEF}(u{:}t) \wedge Q}{P\ [\![\text{Eval } u{:}t]\!]\ Q} \qquad \text{(E1)}$$

$$\frac{P\ [\![\text{Locate } u]\!]\ \text{TAG}(u){=}t \wedge Q}{P\ [\![\text{Locate } u{:}t]\!]\ Q} \qquad \text{(L2)}$$

3. Data Representation Mappings

This section develops the idea that it is sometimes useful to have an efficient mapping between arbitrary types. Specifically, we propose two new operators: LOWER:t(x), a one to one mapping of Pascal values of type t into boolean arrays of sufficient size, and LIFT:t(y), the inverse mapping. The particular mapping used will be implementation dependent. The length of the array in the result of LOWER:t(x) will be given in each implementation by the expression SIZE(t). Some programs using LIFT and LOWER can be written with knowledge of the sizes of the types but without any dependence on the particular mapping used. For instance, conversion of an arbitrary type to boolean arrays of a fixed size could be used in a way similar to Concurrent Pascal's universal parameters, for implementing read and write procedures in operating systems. Other applications may depend on detailed knowledge of the mapping; such programs will not be portable, but we will have techniques for showing that they are free from runtime errors.

Additional applications in systems programming involve the need to convert between addresses and pointers, for instance, in a storage allocator written in a high level language, or in a linking loader for a system in which a program is represented as a pointer to code. To relocate code, it may necessary to convert between a format used for storage, such as arrays of integers, and the machine dependent instruction format. This can be done efficiently if one has knowledge of the mapping implemented by LIFT and LOWER. There are many additional applications involving instruction formats in operating systems. For instance, it is common for hardware input–output devices to depend on control words which must be constructed dynamically. These have formats with integer or character valued fields, for example.

A straightforward extension of our presentation of LIFT and LOWER would be to allow the programmer to declare certain properties of the mapping to be used. For instance, in mapping a record with two fields onto bit arrays of size n, one might

specify that the first field should be mapped into bits 1:m and the second field intc m+1:n. These specifications would be represented in the axiomatic definition as additional entry-exit assumptions for the functions LIFT and LOWER. Alternatively, if the mapping is fixed by a language implementation, the details could be formalized and used to give a verification valid for just that implementation.

Some system programming languages such as C [KR78] and BLISS [BLISS] allow unrestricted mapping between different types. In contrast, our approach is intended control access between types to prevent the construction of invalid values. Since all values to be converted must pass through the operators LIFT and LOWER, we can prevent two kinds of conversion errors which are undetectable in less restricted languges:

1) Errors involving an improper storage location. Each implementation of LIFT and LOWER will assure that conversion results are returned in a storage location of the proper size and alignment. Proper alignment is especially important when lifting to produce a pointer result or an integer more than one byte long.

2) Construction of an invalid value in a proper storage location. This error is roughly equivalent to the construction of an uninitialized value which can then be accessed. Our approach is to specify sufficient preconditions for LIFT to assure that the result is always DEF and Inrange*. It will be possible to use the preconditions to verify that programs using LIFT and LOWER are free from runtime errors.

3.1 Axiomatic Theory of LIFT and LOWER

The operators LIFT and LOWER can be added to the error checking semantics by adding some first order axioms. As usual, conditions for error free use of the operators will be expressed by asserting the conditions under which their results are DEF.

LL1) Any value can be lowered, yielding a well defined value.
DEF(x) ⊃ DEF(LOWER:t(x))

LL2) The function LOWER is one to one.
LOWER:t(x)=LOWER:t(y) ⊃ x=y

LL3) If LOWER:t and then LIFT:t are applied to a well defined value, the result is the same value.
DEF(x) ∧ ***Inrange***(x,t) ⊃ LIFT:t(LOWER:t(x))=x

Because LIFT and LOWER are added to the language as functions, they cannot be used to assign an invalid value to a variable. It is syntactically illegal to use a function application in a place where a variable is required, such as on the left side of an assignment. Example:

```
LOWER:t(x)[n] := TRUE;          -- unsyntactic.
```

The permitted manipulations involving a type t must at some point use LIFT:t, whose precondition ensures that the values are meaningful.

```
barray := LOWER:t(x);
barray[n] := TRUE;
x := LIFT:t(barray);            -- checks value of barray.
```

3.2 Universal Value Parameter

Since all types having the same internal size can be lowered to a common boolean array type, an array parameter in a routine can be used as a universal value parameter.

Example: Universal WRITE procedure.

```
CONST n = ...
PROCEDURE WRITE(x: ARRAY[1:n] OF BOOLEAN);
      .
      .
      .

TYPE t = ...
VAR x: t;
BEGIN WRITE(LOWER:t(x)); ...
```

Note that the usual compile time type checking will require that SIZE(t) be equal to n.

3.3 An example of direct access to pointers.

There is a well known programming technique for representing doubly linked circular lists, using space for only one pointer in each record. Consider a sequence of records R_i, and for each record R_k set

$$R_k.\text{link} = \text{address}(R_{k-1}) \text{ XOR } \text{address}(R_{k+1}),$$

where XOR is bitwise exclusive or.

Now if we are accessing R_k and have the address of R_{k-1}, we can compute the address of R_{k+1} by XORing the two addresses, and similarly, from R_k and R_{k+1}, it is possible to get back to R_{k-1}.

The following program fragment illustrates the use of LIFT and LOWER to implement this XORed pointer representation. A record type REC is declared having a .LINK field of type BITS, an array of booleans large enough to store the result of lowering a pointer. XOR is defined to operate on the boolean arrays. Its definition is not shown here, but its specifications are used in verifying the program fragment.

In the program, the variables p1, p2, and p3, are first set to point to new records. Then each record is linked to the other two creating a circular list. Finally, LIFT is used to move left from p1 giving p3 and right from p1 giving p2. A final assertion in the program states that the new pointers created while moving have the correct values. The program and final assertion can be verified using axioms LL1-3.

```
TYPE
      ptr=frec;
      bits=ARRAY[1:SIZE(ptr)] OF BOOLEAN;
      rec=RECORD info:t; link:bits END;

VAR p1,p2,p3,l,r:ptr;

FUNCTION xor(a,b:bits):bits; external;
      % specifications of xor:
            xor(a,b)=xor(b,a)
            xor(a,xor(a,b))=b      %

BEGIN
NEW(p1);       % allocate 3 RECs %
NEW(p2);
NEW(p3);
            % set up circular list %
p1.link := xor(LOWER(p3), LOWER(p2));
p2.link := xor(LOWER(p1), LOWER(p3));
p3.link := xor(LOWER(p2), LOWER(p1));

            % set l to left link of p1 %
l := LIFT:ptr(xor(p1.link, LOWER(p2)));

            % set r to right link of p1 %
r := LIFT:ptr(xor(LOWER(p3), p1.link));

            % check links %
ASSERT l=p3 ∧ r=p2;
END;
```

The critical part of this program is verifying that the arguments to LIFT are in the image of type ptr. In a language implementation without automatic garbage collection, a pointer value created by a NEW statement remains an element of the type unless it is explicitly deallocated. Thus after the three NEW statements, the values p1, p2 and p3 are all DEF. Using the specifications of xor, it can be shown that the

arguments given to LIFT are equal to LOWER(p3) and LOWER(p2). This satisfies the precondition for definedness of LIFT:ptr in LL3.

Chapter 3. An Example of Verification with Runcheck

This chapter illustrates the actual process of verifying a program of moderate size with Runcheck. The program plays the game of Kalah with the computer acting as board and scorekeeper. Because the program was written for actual use instead of for purposes of illustration, it initially presented various difficulties for verification. We will discuss some small modifications that were made to simplify the verification, and the actual sequence that was followed of assigning assertions, using the verifier, and gradually filling in and correcting the assertions until the absence of runtime errors was verified for the entire program.

The full process of verifying the program will be emphasized instead of simply presenting the final result, for two reasons. First, the example conveys a sense of the amount of effort required to verify shallow properties of moderate sized programs. We would also like to show that verification should not be considered a totally separate activity to be undertaken only after a final version of the program has been written. Attempting to specify and verify a program often leads to a clearer understanding of its structure. Discovering that there are difficulties in specifying or verifying part of a program can help a programmer to improve the clarity of the program.

Here is a sample run of the program (inputs typed by the user are underlined):

```
.RUN KALAH
KALAH  -  TYPE 'H' FOR HELP
      3   3   3   3   3   3
  0                           0
      3   3   3   3   3   3
TOP PLAYS H
KALAH - AN ANCIENT GAME OF AFRICA AND THE MIDDLE EAST.
```

```
        ERS CHOOSE WHO IS TO GO FIRST. THE FIRST PLAYER IS
C    D TOP AND THE SECOND IS CALLED BOTTOM. EACH PLAYER HAS  6
P     AND A KALAH. THE INTEGER ASSOCIATED WITH EACH PIT TELLS
THE NUMBER OF STONES IT CONTAINS. EACH PLAYER IN TURN CHOOSES A
PIT BY ENTERING THE NUMBER OF THE PIT GIVEN BELOW. THE STONES IN
THE PIT ARE DISTRIBUTED TO EACH PIT IN A COUNTER-CLOCKWISE
DIRECTION. IF THERE ARE ENOUGH STONES TO GO BEYOND YOUR KALAH,
THEY ARE DISTRIBUTED TO YOUR OPPONENT'S PITS. IF THE LAST STONE
LANDS IN YOUR KALAH, YOU GET ANOTHER TURN. IF THE LAST STONE
LANDS IN AN EMPTY PIT ON YOUR SIDE, YOU CAPTURE ALL OF THE
OPPONENT'S STONES IN THE OPPOSITE PIT AND ALL STONES INVOLVED
ARE PLACED IN YOUR KALAH.

THE GAME ENDS WHEN ALL THE PITS ON ONE SIDE ARE EMPTY. THE
OTHER PLAYER ADDS THE REMAINING STONES TO HIS KALAH. THE
WINNER HAS THE GREATEST NUMBER OF STONES AND IS AWARDED THE
DIFFERENCE BETWEEN HIS STONES AND HIS OPPONENT'S TOWARDS A ROUND.
THE FIRST PLAYER TO GET  18 WINS THE ROUND. THE LOSER CHOOSES
WHO GOES FIRST IN THE NEXT GAME.

T = TOP;  B = BOTTOM;  K = KALAH

      1T  2T  3T  4T  5T  6T
  KT                          KB
      6B  5B  4B  3B  2B  1B

       3   3   3   3   3   3
   0                          0
       3   3   3   3   3   3
```

Top plays first, moving 3 so that the last stone lands in his kalah, giving him another turn. He then plays 6, capturing the stones in bottom's third pit.

```
TOP PLAYS 3
       4   4   0   3   3   3
   1                          0
       3   3   3   3   3   3

TOP PLAYS 6
       4   4   0   4   4   0
   5                          0
       3   3   0   3   3   3
```

Bottom then plays 2, dropping a stone in his kalah and wrapping around to leave the last stone in top's 6.

```
BOTTOM PLAYS 2
       4   4   0   4   4   1
   5                          1
       3   3   0   3   0   4
```

```
TOP PLAYS 3
T = TOP; B = BOTTOM; K = KALAH
      1T  2T  3T  4T  5T  6T
  KT                        KB
      6B  5B  4B  3B  2B  1B
       4   4   0   4   4   1
   5                        1
       3   3   0   3   0   4
```

Top then mistakenly enters 8, an illegal move because it is an empty pit, and the program reminds him of the positions. The game continues until all of top's pits are empty, and then the program prints out the score.

```
SCORE - TOP  14  BOTTOM  22 - BOTTOM WINS BY   8
EXIT
```

1. Initial preparation

The first step in verifying the program was to read it through, looking for syntax changes needed for it to be accepted by the verifier. The program had been written in standard PDP-10 Pascal, and the language accepted by the verifier [SVG79] has a number of additional restrictions. For instance, type checking is very strict in the READ and WRITE procedures. In the verifier version of the program, device TTY was declared to be a file of integers, and a large number of WRITE statements for printing strings (such as the program's help instructions) had to be removed. It is also necessary in the current verifier to list explicitly, for each procedure, the readonly or read-write global variables.

Some of the restrictions in the verifier are present to insure that the complete effect of every program can be captured within the VCG semantics. In some cases this may have made the verifier too restrictive. The alternatives to the current situation are either to develop full semantic definitions for some aspects of Pascal not now permitted, or to use intentionally weak semantics, permitting some operations such as terminal output to appear in programs without fully defining their effects.

At this point, two other small changes were made in the program text to simplify verification. One of the changes was to remove aliased variables from a procedure call — the procedure call rules in the verifier do not permit aliasing, although in principle more general rules could be developed. Depending on whether or not one allows aliasing, there may be a trade off between the conciseness and efficiency of programs, and the complexity of specification and verification. The example which we will consider shows that aliasing can add difficulty to understanding and verifying a program even if it is permitted by the procedure call rule. In the process of studying the effect of aliasing, a cleaner way of organizing part of the program was discovered.

Another change was made in the program only to simplify the verification. The original program would have been acceptable, but in order to show absence of runtime errors in one statement, it would have been necessary to perform more detailed verifications of other large portions of the program. It is frequently the case that the correctness of some small portion of a program is dependent on the preservation of a global property by many other portions. In such cases it is often better, both from the standpoint of verification and that of good programming practice, to consider modifying the program to eliminate the unnecessary dependency.

2. A look at Aliasing

In the program, each player's row of pits is represented as a variable of type SIDE = ARRAY [0 . . PITCOUNT] OF INTEGER, where PITCOUNT is the number of pits for each player, and the zero position is the Kalah. The state of the game is maintained in the global variables TOP and BOTTOM of type SIDE:

```
CONST
 PITCOUNT = 6; % NUMBER OF PITS FOR EACH PLAYER. (NORMALLY 6) %
 STONES = 3; % STARTING NUMBER OF STONES PER PIT. (NORMALLY PITCOUNT/2) %

TYPE
 POSITION = 0:PITCOUNT;
 SIDE = ARRAY [POSITION] OF INTEGER;

VAR
 TOP, BOTTOM: SIDE;
```

A procedure PRINTBOARD is called to print out the current state of the board. Note that TOP and BOTTOM are referenced as read-only globals.

```
 PROCEDURE PRINTBOARD;
 VAR
  PIT: POSITION;
 BEGIN
  WRITE(TTY,'    ');
  FOR PIT := 1 TO PITCOUNT DO
   WRITE(TTY,TOP[PIT]: 4);
  WRITELN(TTY);
  WRITE(TTY,TOP[0]: 4);
  FOR PIT := 1 TO PITCOUNT DO
   WRITE(TTY,'    ');
  WRITELN(TTY,BOTTOM[0]: 4);
  WRITE(TTY,'    ');
  FOR PIT := PITCOUNT DOWNTO 1 DO
   WRITE(TTY,BOTTOM[PIT]: 4);
  WRITELN(TTY);
  WRITELN(TTY)
 END; % PRINTBOARD %
```

Dividing the board up into TOP and BOTTOM poses a problem when it comes to writing the part of the program for moving the stones. Moves that wrap around the

end or capture stones require access to both sides of the board. It is inconvenient to refer to the sides as TOP and BOTTOM in these parts of the program; one wants instead to refer to the sides as the side making the current play and the opposite side. In the program, this is accomplished by calling a procedure PLAY to rebind the two sides to the variables US (whichever side is currently moving) and THEM (the other side).

```
PROCEDURE PLAY(VAR US, THEM: SIDE; TOPPLAY: BOOLEAN);
% CALLED FOR EACH PLAY. RETURNS FALSE WHEN ONE PLAYERS TURN ENDS. %
```

Ideally, in this plan for the program, the procedure PLAY should be symmetric between the two sides, referring to them only by the names US and THEM. The effect of calling PLAY would then depend only on the values of the arguments, and not on their names. This plan was not carried out fully. PRINTBOARD is called within PLAY, and since TOP and BOTTOM are globals of PRINTBOARD, they are globals of PLAY. In the procedure calls PLAY(TOP,BOTTOM,TRUE) and PLAY(BOTTOM,TOP,FALSE), TOP and BOTTOM become aliases with US and THEM.

The ability to refer to a variable by different names leads to programs that are concise and efficient, but difficult to understand and specify.[1] It is often the case that a procedure which will be called with aliasing cannot be understood from its text alone — one is forced to look outside to the caller. In reading and understanding the text of a procedure, aliasing is an exceptional case; one tends to think of each identifier as a distinct variable. Aliasing lends itself to misunderstanding not so much because it introduces complexity, but because (at least in current programming languages) the complexity is concealed.

Here is an outline of the nesting of variable and procedure declarations in which the aliasing occurs:

[1] EQUIVALENCE statements in FORTRAN are an extreme example.

```
VAR TOP, BOTTOM:SIDE;

PROCEDURE PRINTBOARD;
 BEGIN . . . END;     % (refers to TOP and BOTTOM) %

PROCEDURE PLAY(VAR US, THEM:SIDE; TOPPLAY:BOOLEAN);
  . . .
    PROCEDURE READMOVE
      . . .
        REPEAT
          PRINTBOARD;
          IF TOPPLAY THEN WRITE(TTY, 'TOP PLAYS ');
            ELSE WRITE(TTY, 'BOTTOM PLAYS ');
          . . .
      . . .
    END;
  . . .
END;

BEGIN  % Main Routine %
  . . . PLAY(TOP,BOTTOM,TRUE); . . . PLAY(BOTTOM,TOP,FALSE); . . .
END;
```

The first thing to notice is that because TOP and BOTTOM are always passed as globals, there is no direct indication in the text that they are referenced in PLAY. One can discover that the variables are used only by noting the call on PRINTBOARD and referring back to its definition. The next point of difficulty in understanding that is likely to occur while reading the text of PLAY is that one may notice that TOP and BOTTOM are referenced as globals but not changed, and mistakenly infer that the values of TOP and BOTTOM seen by PRINTBOARD are the initial values from the time when PLAY is entered. Of course this is not the case, but to understand, one would have to read the main routine and see the aliasing procedure call. The combination of global variables and aliasing encourages the construction of programs in which local details cannot be understood unless one has thoroughly examined the entire program.

If we change PRINTBOARD to take the two sides as parameters instead of as globals, the aliasing in PLAY can be eliminated by making the call on PRINTBOARD conditional.

Notation: in this chapter, the original program is displayed in upper case, all changes and additions for verification are shown in lower case.

```
IF TOPPLAY THEN
     begin printboard(us, them); WRITE(TTY, 'TOP PLAYS ') end
  else begin printboard(them, us); WRITE(TTY, 'BOTTOM PLAYS ') end;
```

The conditionality was implicit before in the pattern of aliasing; this version of the program makes it explicit. Some of the complexity of the program has been transferred from variable bindings to an explicit test, with a small cost in execution time.

The new version can be more readily understood and since it performs the same function, its specifications should be no more complicated than the original's. In order to specify the original program, it would have been necessary to describe in some way the functional dependence on the names of the parameters achieved by aliasing. The new version has the advantage that it can be described independently of the names of the actual parameters.

Explicitly writing out the arguments to PRINTBOARD calls attention to the functional messiness of PLAY and READMOVE. The program could be further improved by separating the operations of printing the board and announcing the current player from the operations of reading a move and changing the board. The printing operations are based on the identification of the sides as TOP and BOTTOM, while the reading and moving operations are symmetric and the procedures for them are clearer are more efficient without references to TOP and BOTTOM.

3. When to leave a potential error

A procedure NOROCKS is called after each move to test whether all of the pits on one side have become empty, indicating that the game is finished. The WHILE loop in the procedure NOROCKS presents a typical difficulty: the index PIT can become negative, giving a subscripting error, if the parameter US is an array of all zeros. The actual parameters supplied to NOROCKS are always one of the sides, TOP and BOTTOM. Since the zero position in a side is the Kalah, it is not possible for all the entries in US to be zero: the rules of the game as enforced by the program make it impossible for one player to capture all of the stones, and so if all of the pits on a side are empty, there must be stones in the Kalah.

```
PROCEDURE NOROCKS(VAR US: SIDE);
% TESTS FOR THE TERMINATION CONDITION OF THE GAME. %
VAR
  PIT: POSITION;
BEGIN
  PIT := PITCOUNT;
  WHILE (US[PIT]=0) DO
    PIT := PIT - 1;
  IF PIT = 0 THEN BEGIN
    TURNDONE := TRUE;
    GAMEOVER := TRUE
  END
END; % ROCKS %
```

In order to verify the necessary entry condition on US, it would be necessary to invent an invariant for the sides, and show that it is maintained throughout the program whenever one of the sides is changed. This verification is quite feasible, but requires much more detail than usual. There are a number of alternatives; the question becomes whether the detailed verification is worthwhile. This in turn depends on one's reason for verifying the program. For this illustration, we chose to assume that the verification was mainly intended to assure absence anomalies that could produce runtime errors. Given this limited purpose, a reasonable way to proceed was to modify the test of the WHILE loop to assure absence of runtime errors locally,

regardless of the value of US, while maintaining functional equivalence under the assumption that US would always have a non-zero element.

```
WHILE (US[PIT]=0) and (pit>0) DO   PIT := PIT - 1;
```

Changing the program is consistent with the belief that verification of shallow properties is not intended to give an absolute guarantee of correctness, but rather to extend the range of mechanical checking performed on a program. The program must not be regarded as a sacred, immutable text carved in stone, to be verified and then pronounced infallible. The verifier is a tool for programming; it will be used to the extent that it helps to reduce the total amount of effort needed to produce high quality programs. Since we are not attempting to verify the detailed properties of the Kalah program the programmer must still attempt to make it work correctly by the usual methods. Using a minimum of effort, Runcheck will show the absence of a potentially large number of common problems which cannot be detected during compilation.

If one is unsure that the assumption about US will be maintained, it may be better to test the assumption at runtime and abort the program if an error is detected. Failure of the assumption indicates a major flaw in the operation of the program, which could be masked by strengthening the WHILE test. This approach explicitly leaves open the possibility of an error in one statement. Verification is still of great value with this approach, because all of the other possible runtime errors have been eliminated, and the remaining one can be tested at runtime at a small cost.

```
WHILE (US[PIT]=0) DO
 begin
  testassertion pit>0;
  PIT := PIT - 1;
 end;
```

4. Initial assignment of assertions

After modifying the program, each procedure was examined and a trial set of entry and exit assertions was written. It is not necessary (or usually possible) for the assertions to be exactly right at this stage. From experience, it seems best to assign assertions fairly quickly and then use the verifier as a guide for filling in whatever is missing.

```
% KALAH - AN ANCIENT GAME OF AFRICA AND THE MIDDLE EAST. %
% JOHN RAMSDELL  DEC 1979 %

CONST
  PITCOUNT = 6; % NUMBER OF PITS FOR EACH PLAYER. (NORMALLY 6) %
  STONES = 3; % STARTING NUMBER OF STONES PER PIT. (NORMALLY PITCOUNT/2) %
% A MORE INTERESTING GAME FOR EXPERTS RESULTS BY SETTING STONES %
% TO A VALUE BETWEEN PITCOUNT/2+1 AND PITCOUNT. %
TYPE
  POSITION = 0:PITCOUNT;
  SIDE = ARRAY [POSITION] OF INTEGER;
VAR
  tty: file of integer;
  PIT: POSITION;
  TOP, BOTTOM: SIDE;
  GAMEOVER: BOOLEAN;

  PROCEDURE PRINTBOARD(top,bottom:side);
  global(var tty);
  entry def(top)∧def(bottom)∧def(pitcount);
  exit true;
  VAR
    PIT: POSITION;
  BEGIN
%   WRITE(TTY,'    '); %
    FOR PIT := 1 TO PITCOUNT invariant true DO
      WRITE(TTY,TOP[PIT]);
%   WRITELN(TTY); %
    WRITE(TTY,TOP[0]);
%   FOR PIT := 1 TO PITCOUNT DO
      WRITE(TTY,'    ');%
    WRITE(TTY,BOTTOM[0]);
%   WRITE(TTY,'    ');%
    FOR PIT := PITCOUNT DOWNTO 1 invariant true DO
      WRITE(TTY,BOTTOM[PIT]);
%   WRITELN(TTY);%
```

```
%   WRITELN(TTY);%
 END; % PRINTBOARD %

 PROCEDURE HELP;
 exit true;
 BEGIN
%      (prints instructions shown in the sample protocol)     %
 END; % HELP %

 PROCEDURE HELPMOVE;
 global(var tty);
 entry def(pitcount);
 exit true;
 VAR
   PIT: POSITION;
 BEGIN
%   WRITELN(TTY);%
%   WRITELN(TTY,'T = TOP; B = BOTTOM; K = KALAH');%
%   WRITELN(TTY);%
%   WRITE(TTY,'    ');%
   FOR PIT := 1 TO PITCOUNT invariant true DO
     WRITE(TTY,PIT %, 'T'%);
%   WRITELN(TTY);%
%  WRITE(TTY,'  KT');%
   FOR PIT := 1 TO PITCOUNT invariant true DO
     WRITE(TTY, 0 %'  '%);
%   WRITELN(TTY,'  KB');%
%   WRITE(TTY,'    ');%
   FOR PIT := PITCOUNT DOWNTO 1 invariant true DO
     WRITE(TTY,PIT %, 'B'%);
%   WRITELN(TTY);%
%   WRITELN(TTY)%
 END; % HELPMOVE %

 PROCEDURE PLAY(VAR US, THEM: SIDE; TOPPLAY: BOOLEAN);
 % CALLED FOR EACH PLAY. RETURNS FALSE WHEN ONE PLAYERS TURN ENDS. %
 global(var tty,gameover);
 entry def(us)∧def(them)∧def(pitcount);
 exit def(us)∧def(them)∧def(gameover);
 VAR
   PIT: POSITION;
   LASTPIT, STONES: INTEGER;
   TURNDONE: BOOLEAN;

   PROCEDURE READMOVE(VAR PIT: POSITION);
   global(us,them,topplay;var tty);
   entry def(us)∧def(them)∧def(topplay)∧def(pitcount)∧0<pitcount;
   exit def(pit)∧0<pit∧pit<=pitcount;
```

```
 VAR
  GOODMOVE: BOOLEAN;
  NUM: INTEGER;
 BEGIN
  GOODMOVE := FALSE;
  REPEAT
   IF TOPPLAY THEN
    begin
    printboard(us,them);
    WRITE(TTY,0 %'TOP PLAYS '%)
    end
   ELSE
    begin
    printboard(them,us);
    WRITE(TTY,1 %'BOTTOM PLAYS '%);
    end;
   READ(TTY,NUM);
   IF NUM > PITCOUNT THEN
    HELP
   ELSE IF NUM > 0 THEN
    IF US[NUM] <> 0 THEN BEGIN
     PIT := NUM;
     GOODMOVE := TRUE
    END;
   IF NOT GOODMOVE THEN
    HELPMOVE
  UNTIL GOODMOVE invariant true
 END; % READMOVE %

 FUNCTION MODULUS(NUMBER, BASE: INTEGER): INTEGER;
 entry base=>1;
 exit 0<=modulus ∧ modulus<=base-1;
 BEGIN
  IF NUMBER => 0 THEN
   MODULUS := NUMBER MOD BASE
  ELSE BEGIN
   REPEAT
    NUMBER := NUMBER + BASE;
   UNTIL NUMBER => 0
   invariant number<=base-1;
   MODULUS := NUMBER
  END
 END; % MODULUS %

 PROCEDURE MOVE(VAR US, THEM: SIDE; PIT: POSITION);
 global(var stones);
 entry def(stones)∧1≤stones∧def(pitcount)∧def(us)∧def(them)
   ∧def(pit)∧0<=pit∧pit≤pitcount;
```

```
 exit def(us)∧def(them)∧def(stones);
 VAR
  INDEX: POSITION;
  SMALL: INTEGER;
 BEGIN % DISTRIBUTES STONES TO THE PITS. %
  STONES := STONES - PIT - 1;
  SMALL := -STONES;
  IF SMALL < 0 THEN
   SMALL := 0;
  FOR INDEX := PIT DOWNTO SMALL invariant true DO
   US[INDEX] := US[INDEX] + 1;
  IF STONES > 0 THEN
   MOVE(THEM, US, PITCOUNT)
 END; % MOVE %

 PROCEDURE NOROCKS(VAR US: SIDE);
 % TESTS FOR THE TERMINATION CONDITION OF THE GAME. %
 global(var turndone;var gameover);
 entry def(us)∧def(pitcount)∧def(turndone)∧def(gameover);
 exit def(us)∧def(turndone)∧def(gameover);
 VAR
  PIT: POSITION;
 BEGIN
  PIT := PITCOUNT;
  invariant 0<=pit∧pit<=pitcount
  WHILE (US[PIT]=0) and (pit>0) DO
   PIT := PIT - 1;
  IF PIT = 0 THEN BEGIN
   TURNDONE := TRUE;
   GAMEOVER := TRUE
  END
 END; % ROCKS %

BEGIN % PLAY %
 REPEAT
  READMOVE(PIT);
  % THE STONE THAT MOVED THE FURTHEST ENDS UP IN LASTPIT. %
  LASTPIT := MODULUS(PIT - US[PIT], 2 * PITCOUNT + 2);
  STONES := US[PIT];
  US[PIT] := 0;
  MOVE(US, THEM, PIT - 1); % MOVE STONES TO NEW PITS. %
  TURNDONE := TRUE;
  IF LASTPIT = 0 THEN
   TURNDONE := FALSE % REPLAY IF LAST STONE ENDS IN KALAH. %
  ELSE IF LASTPIT <= PITCOUNT THEN
   IF US[LASTPIT] = 1 THEN
    IF THEM[PITCOUNT + 1 - LASTPIT] <> 0 THEN BEGIN
     % CAPTURE OPPONENTS STONES. %
```

```
        US[0] := US[0] + THEM[PITCOUNT + 1 - LASTPIT] + 1;
        THEM[PITCOUNT + 1 - LASTPIT] := 0;
        US[LASTPIT] := 0
       END;
    % TEST FOR END OF GAME. %
    NOROCKS(US);
    NOROCKS(THEM)
  UNTIL TURNDONE invariant true
 END; % PLAY %

entry true;
exit true;
BEGIN % KALAH %
 GAMEOVER := FALSE;
 TOP[0] := 0; % INITALIZE GAME. %
 BOTTOM[0] := 0;
 FOR PIT := 1 TO PITCOUNT invariant true DO BEGIN
  TOP[PIT] := STONES;
  BOTTOM[PIT] := STONES
 END;
% WRITELN(TTY);%
% WRITELN(TTY,'KALAH - TYPE "H" FOR HELP');%
% WRITELN(TTY);%
 REPEAT   % PLAY GAME. %
  PLAY(TOP, BOTTOM, TRUE);
  IF NOT GAMEOVER THEN
   PLAY(BOTTOM, TOP, FALSE)
 UNTIL GAMEOVER invariant true;
 % GAME OVER - PRINT FINAL SCORE. %
 FOR PIT := 1 TO PITCOUNT invariant true DO BEGIN
  TOP[0] := TOP[0] + TOP[PIT];
  BOTTOM[0] := BOTTOM[0] + BOTTOM[PIT]
 END;
% WRITELN(TTY);%
% WRITE(TTY,'SCORE - TOP', TOP[0]: 4, ' BOTTOM', BOTTOM[0]: 4, ' - '); %
% IF TOP[0] > BOTTOM[0] THEN
  WRITELN(TTY,'TOP WINS BY', TOP[0] - BOTTOM[0]: 4)
 ELSE IF TOP[0] < BOTTOM[0] THEN
  WRITELN(TTY,'BOTTOM WINS BY', BOTTOM[0] - TOP[0]: 4)
 ELSE
  WRITELN(TTY,'NO WINNER')
%
END.
```

5. Using Runcheck

5.1 Verifying the program

Initially, several tries were needed before the program with assertions passed all of the verifier's syntax checks. Eventually, the program was accepted, and the verifier produced some additional loop invariants and generated the verification conditions. A large number of the conditions did not simplify to True.

1. The Exit condition for MODULUS was not provable, because the verifier has no built in knowledge of the function MOD, and no inference rules had been given. The following axioms were added for the next try:

$0 \le x \wedge 0 < y \supset 0 \le (x \text{ MOD } y) < y,$
$DEF(x) \wedge DEF(y) \supset DEF(x \text{ MOD } y)$

2. In the procedure PLAY, there were a large number of unproven conditions of the form 0<US[PIT]. It was not immediately clear what had caused this problem, so further consideration was postponed.

3. There were also unproven conditions of the form DEF(GAMEOVER) for PLAY. Looking back at the program listing, it was recalled that the variable GAMEOVER is set to TRUE by the procedure NOROCKS to signal the end of the game. GAMEOVER is initially set to FALSE in the main procedure, and is tested there after each call to the procedure PLAY. In the first assignment of assertions, DEF(GAMEOVER) had been used as an entry and exit assertion for NOROCKS, and as an exit assertion in PLAY, but not as an entry assertion. The unprovable conditions for PLAY resulted from the missing entry condition, which was added for the next try. Since GAMEOVER is assigned to in NOROCKS but never referenced there or in PLAY, it would also have been possible to delete all of the entry and exit assertions for GAMEOVER from the two procedures. The verifier would still be able to prove DEF(GAMEOVER) at the points where it is

referenced in the main procedure, because GAMEOVER is initialized to FALSE and, by the Lessdef lemma, cannot later become uninitialized.

4. For the procedure READMOVE, there were unproven conditions DEF(PIT) and 1≤PIT≤6. These resulted from the exit assertion, which was not provable when leaving the main REPEAT loop, because the invariant had initially been simply set to TRUE. Note that since PITCOUNT was declared a constant, the verifier substituted the value 6 wherever it originally occurred in an assertion. The loop in READMOVE reads numbers from the terminal until a legal move is entered, and then sets PIT to the number read, which must be between 1 and PITCOUNT, and sets GOODMOVE to TRUE. For the next try, the invariant was set to

$$\text{GOODMOVE=TRUE} \supset (\text{DEF(PIT)} \wedge 1 \le \text{PIT} \wedge \text{PIT} \le \text{PITCOUNT}).$$

It is interesting to note in passing that the invariant for PIT cannot be expressed as a conjunction of linear inequalities, because of the dependence on the variable GOODMOVE. Specialized methods for automatically generating linear loop invariants have been studied by some researchers; our experience indicates that non-convex assertions are required with sufficient frequency that a verifier based solely on automatically generated convex assertions, without user assistance, would be of very limited usefulness even for verifying shallow properties such as absence of runtime errors.

After making the changes mentioned above, the verifier was run again, and only the conditions 0<US[PIT] in PLAY remained unproven. Looking at the body of PLAY, it was observed that the variable STONES is assigned the value US[PIT] and then the procedure MOVE is called with an entry assertion containing 1≤STONES. This is how 0<US[PIT] appeared in the VC for PLAY.

On entry to MOVE, STONES is set to the number to stones to be distributed, which must be greater than 0. The entry condition 1≤STONES is always satisfied when

MOVE is called, but looking at the program, it was realized that the assertion should not be needed for proving absence of runtime errors in MOVE. The condition was deleted from the entry assertion, and when the verifier was used a third time, all of the VCs were completely proven.

It would have also been possible to establish the truth of 0<US[PIT] in PLAY by strengthening the exit assertion of the procedure READMOVE, which always sets PIT to a value such that 0<US[PIT] is true.

5.2 Generalizing the verification

Once an initial verification has been obtained, it is sometimes worthwhile to experiment further to see what will happen if some of the initial assumptions are lifted. In the Kalah program, PITCOUNT and STONES are declared as integer constants with values 6 and 3, but a comment in the program suggests using other values for a more interesting game. In order to check for absence of runtime errors for all possible settings, PITCOUNT and STONES were redeclared to be variables in the outermost block. The entire program was then reverified with only the initial assumptions

$$DEF(PITCOUNT) \wedge 1 \leq PITCOUNT \wedge DEF(STONES),$$

showing that PITCOUNT could be any positive constant and STONES could have any value. The effect of this generalization is difficult to achieve by ordinary means such as program testing.

Chapter 4. Verification and the Reliability of Computer Programs

The principles of program verification are now well understood, but what can we say of the practice? That at present we are able to specify and verify small and often very intricate programs. That a few large programs have been specified and verified with a great deal of effort. And that in addition to increasing our confidence in correct programs, experience with the verifier shows it to be extremely helpful for finding errors in programs.

Can verification become a practical tool for increasing the reliability of larger programs? Among the myths which have hindered realistic understanding of this question is the belief that verification can or should somehow attempt to eliminate all programming errors. Verification is expensive and cannot guarantee correctness in any absolute sense. As a practical tool, verification will be used only to the extent that it is a cost competitive way of obtaining a desired degree of reliability.

We say that something is reliable if we can put our trust in it. To decide whether something is reliable, we have to know the ways in which it is likely to fail. For physical objects such as bridges and integrated circuits, reliability can be easily observed and measured: simply use something and wait for it to stop working. For instance, if we wanted to measure the reliability of an integrated circuit, we could operate it under a variety of conditions of voltage, temperature, and vibration, and see how well it performed its intended function. When we speak of the reliability of computer programs, we mean something that is different in an important sense. Computer programs are pure function without materials or assembly which can behave in unpredictable ways. The failure of a program is a failure of functional design; the question of reliability for programs is more closely analogous to the question of whether a circuit performs the proper function under ideal conditions

than the question of reliability of circuits. Correctness of function is much more of an issue for programs than for circuits or bridges because the functions can be so complex. Reliability is a subtle issue for programs because the intended function is often incompletely or incorrectly understood.

One of the remarkable unities in physical science is the applicability of half a handful of probability distributions to a broad range of phenomena. Social scientists have also made use of simple probabilistic assumptions in their models, perhaps with less justification. But we can see no justification for treatments that attempt to apply to software the models of reliability that have been developed for various areas of engineering. Ideas which help us to understand the failures of physical systems such as circuits or bridges will tell us little about design errors in programs.

Because individuals often have incorrect or incomplete ideas about the intended functions of a program, programs used by many people are unlikely to be reliable unless different users can reach a precise agreement that a program fulfills its intended function. In our view, the value of verification is that it helps people to reach very strong and precise agreements about programs. The nature of such an agreement, which we will simply call a *consensus*, can be best appreciated through a detailed, pragmatic examination of the verification process.

In the remainder of this chapter we will draw on experience with Runcheck and the Stanford Pascal Verifier to clarify several practical issues:

1) How verification contributes to reliability even in the absence of absolute correctness.

2) What kinds of applications of program verification appear to be feasible for large programs.

3) How verification can be combined with other methods such as testing.

While knowledge of the theory underlying the formal operations of verification is now widespread, the human aspects of verification, such as the amount of labor required and the effects of human errors, have been rarely discussed. We hope that the observations in this chapter will contribute to more realistic understanding of verification.

Throughout this chapter we will be developing a new view of the meaning of program verification. The classical view has been that verification sought to assimilate programming into formal mathematics, thereby elevating it above uncertainty. Our new view emphasizes the use of mathematical methods to reach a consensus, or strong agreement among users about the correctness of a program. Of course, there are many methods, including testing and informal design reviews, which can give some degree of confidence in programs. But we view verification as a tool which can be used to form a stronger consensus than would be otherwise possible.

1. Program Specification and Consensus

Implicit in the notion of reliability is the view that it is not sufficient for a program to faithfully perform some function unless that function can be well understood by users. It is essential for the creators of a reliable program to communicate its function through precise and understandable documentation.

One of the most frequently raised criticisms of program verification is that the true objectives of a program are usually known only informally to the programmer, while to apply program verification, it is neccessary to develop formal program specifications. Since these specifications may be in error, or may inaccurately or incompletely reflect the programmer's informal intentions, program verification cannot give absolute assurance that intentions will be fulfilled. In our view, there is some validity to this criticism: the problems of formalizing specifications are not to be trivialized and no mathematical procedure can demonstrate the consistency of informal ideas. But it would be naive to infer directly from this argument, as some critics have, that program verification is doomed to be unable to increase our confidence in and the reliability of programs.

If there is one thing that is common to all of science and engineering, it is use of formal mathematical methods to investigate informal intuitions and intentions. Science and engineering derive their power from transitions of informal ideas to precise mathematical descriptions. Until formalized, a theory can never be subjected to critical scientific analysis. Similarly, the effort to make sure that computer programs accomplish a desired objective must be based on precise, understandable descriptions of the purpose. If we are truly unable to make a precise, understandable statement of the purpose of program, how likely is it that the program can ever be reliable?

When can we say that a computer application performs the desired function? If just

one person conceives the function, his mental concept may be incomplete or not in agreement with other people. So it is best to say that a number of people should agree that the function is the right one. Given a set of specifications, different people can study it and attempt to reach agreement that it corresponds to the informal notions each of them holds. Without precise specifications, what methods are there to reach such an agreement? To have a number of different people attempt to review a very large program without precise specifications is impractical and unlikely to give strong assurance. (Although review of a program may be a helpful step in formulating precise specifications.) In programs, it is difficult to suppress details which distract from readability. More important, specifications can be structured for ease of understanding instead of effective, efficient execution.

We do not mean do suggest that it is possible in every case for the purpose of a program to be stated precisely. On the contrary, there may be programs which, for instance, attempt to compose music or amusing anecdotes, for which no precise statement of purpose can exist. Different people, by testing such a program very thoroughly, could come to some agreement among themselves that on the examples they have seen, the program does accomplish an informally stated purpose. In such a case, the testers might in fairness reach a concensus that the program is artistically talented. But even if such an agreement has been reached, a program of this type cannot be said to belong to the category of reliable software, because by testing alone against an ill defined set of criteria, we can develop no strong assurance that the next composition generated by the program will not be found to be unmusical or unamusing. The point of this example is that there are limits to the notion of reliable software, that some programs can be useful without being reliable. But if the purpose of a program cannot be stated with sufficient precision for people to reach a consensus, it is hard to see how it can be reliable.

What about informal specifications? They are certainly useful for many purposes. Specifications can be shortened in two ways: by referring to definitions that are

generally known, trusting each reader to have the same understanding, or by becoming less precise. To the extent that "informal" means "imprecise," informal specifications will be unable to contribute to reliability.

Finally, we have to say a few words about a second myth of verification: that specifications should always describe the program completely. There is a common criticism of verification which goes, "There are things in many programs which are hard to specify independently and completely. Therefore verification cannot contribute to reliability." Runcheck is based on the idea of verifying very incomplete specifications — only enough to show absence of runtime errors. We feel that, contrary to myth, many other important applications of verification will depend on partial specifications which can be written and checked relatively easily.

2. Concerning False Proofs

One of the central arguments against the effectiveness of program verification is that individual verifications are unlikely to command the attention of a large critical audience, and therefore errors in proofs are unlikely to be detected [DLP79]. In our view, there can be no absolute assurance of the validity of proofs. However, a close look at the theory and construction of a verifier (Stanford Pascal Verifier or Runcheck) will show that it has the potential to be at least as reliable as any other stable, widely used piece of software. Whatever errors may happen to be in a verifier are very likely to be detected as it is used, resulting in a stable, reliable system even in the absence of a verification of the verifier.

One may ask what the value of verification is if the reliability of the verifier is not of a fundamentally different nature than that of other reliable programs. Over the course of time, a verifier concentrates the experience of its designers and users and the users of verified programs. Faults which are discovered can be corrected, and so will not affect later users.

A verifier is the center of a consensus between people who propose verification methods, implementors of the verifier, its users, and the users of verified programs. Any fault in the verifier is observable by one or more groups (more about this later), and can then be corrected. The process we are describing is a familiar one: verification is the application of the scientific method to the field of programming. The ultimate source of the verifier's reliability is not some set of absolute truths, but rather the process by which scientific theories are validated. At any time, a proof produced by the verifier should represent our best thinking about what constitutes a valid proof. New users are spared from repeating old mistakes.

The advantage given by the verifier is that the experience concentrated in it can be applied in one shot to new programs. If a new program is heavily used and carefully

maintained over a long period of time, it can reach a high level of reliability without having been verified. But the verifier helps us to reach this result much more quickly. Of course, this depends on the availability of the proper specifications. To reiterate our previous comments, 1) partial specifications (e.g. specifying that a program should be free from runtime errors) are often the most practical, and 2) if we really do not know how to specify some aspect of a program, there are strong grounds for believing that the program cannot possibly be reliable.

In the remainder of this section, we will discuss the reliability of the major components of a verifier. For concreteness, we will consider the Stanford verifier, but our comments also apply to Runcheck, which is a version of the Stanford verifier, and to verifiers in general.

The three main components of the Stanford verifier are a parser, much like the front end of a compiler, the verification condition generator (VCG), which implements the semantic definition of Pascal, and the theorem prover, which is independent of the programming language accepted by the verifier.

The parser component consists of a table driven context free parser and semantic routines which use a symbol table to perform the usual semantic checking performed by compilers. None of this is new technology; the parser is very reliable because standard compiler construction is now routine.

VCG converts the parse tree of the program with assertions into a set of first order formulas whose validity implies the consistency of the program and assertions. The question of VCG's reliability reduces to two separate issues. One is whether there is a problem in the axiomatic definition; the other is the correctness of the VCG implementation. Roughly speaking, the ultimate question relating to the soundness of the axiomatic definition is whether the intuitive semantics of Pascal is a model of the formal definition. There have been formal demonstrations of consistency between the

axiomatic definition and other semantic definitions,[1] but consistency proofs cannot completely resolve the issue of whether a formal semantics corresponds to the intuitive semantics. Fortunately, the language Pascal and its axiomatic definition have received widespread attention. The existence of a body of published literature correcting and refining the original definition is evidence of the formation of a strong consensus about the definition's correctness.

An interesting question beyond the scope of this thesis is: How does one account for the existence of agreement on the intuitive semantics of Pascal or other programming languages? One could name languages for which agreement would be much less certain. Looking for answers close at hand, one finds the factors of clean language design, and the conservatism of the language — its dependence on only previously well known concepts. The same considerations apply to the understandability of programs in general. One can only speculate about deeper explanations. Perhaps the experience with programming languages can tell us something about language itself. The commonly observed tendancy of programming languages to guide and constrain thought may indicate that we use programming languages as languages in some sense ([Wh56], Part 4 [We71]). Perhaps the process of acquiring the general rules of a programming language from fragmentary explanations and examples is related to (and can be explained in terms of) the process of learning a language.

One of the most controversial aspects of verification has to do with the fact that sometimes it is difficult to formally define part of a language. When this happens, there are several possibilities. It may be that we simply don't know enough about how to give a concise definition, and that further research could find ways of doing it. It may be that the feature represents poor language design: inherently difficult to describe and understand. The third possibility is that the feature represents a desirable form of complexity in the language. Complexity is not undesirable per se,

[1] If we choose to regard a compiler as a kind of formal language definition, then proofs of compiler correctness can be viewed as another form of consistency proof.

and there is a trade off between the expressive power of a language and the complexity of programs written in the language.

If we cannot give a concise, understandable definition of part of a language, it will probably be difficult to build all of the tools needed for programming — not just the verifier, but compilers, optimizers, debugging tools, and program analyzers of all kinds. Even if these tools can be constructed, they may be less reliable, because it will be more difficult for implementors to understand and agree on the semantics. So difficulty in formal definition is a warning that there may be further troubles with a language design. Formal definability gives the langauge designer another way to test a design, in addition to the usual ways based on experience with other languages and difficulty of implementation, etc. Designs can contain unpleasant surprises — combinations of features that interact in unexpected ways. When designs are judged solely by their intuitive semantics, the surprises can remain hidden, because intuitive semantics tend to be incomplete.

The most difficult issue in the soundness of VCG is the language definition, but a little should be said about the VCG implementation. For the most part, VCG is a straightforward translation of the axiomatic definition into operational form. The actual program for VCG could be generated automatically from a table of the inference rules. It would not be difficult to verify that formulas constructed by VCG correspond exactly to the axiomatic definition.

Different considerations affect the soundness of the first order theorem prover. The concept of soundness is readily specified, but the theorem prover employs novel and somewhat complicated algorithms. The implementation is more complicated than it might be if efficiency was not of prime importance. A major part of the current theorem prover is a program for determining the satisfiability of a set of linear inequalities using the simplex algorithm. Here, the algorithm is well known but its implementation is rather complicated. In general, verification of the theorem prover

is somewhat beyond the current state of the art, but since the specifications are not difficult, it may eventually be possible to verify much of it.

At present, the reliability of the theorem prover rests largely on the alertness of users in reporting problems. Herein lies a serious misunderstanding on the part of verification critics. Among those who have not used a program verifier, there is belief in a third myth of verification: that one simply submits a program and waits until the verifier responds with either VERIFIED or NOT VERIFIED, i.e. that the system does all the work by itself and that there is no reason for its results not to be accepted uncritically. In reality, any new nontrivial verification is an undertaking in which the user must interact with the verifier to produce a proof. In the course of the interaction, a program is usually submitted many times with different documentation. After each run, the user examines the output of the theorem prover in the form of partially simplified verification conditions and summaries of the steps taken in the proof. The user must study these results closely; they contain the clues needed for understanding why the proof was not completed. Analysis of simplified verification conditions is a special skill that one must learn in order to use the verifier. In the process of analyzing a VC, one notes which consequents were provable and which were not, and one must understand how the documentation in the program was used to prove one formula but why it did not suffice to prove some other. Implementation errors in the theorem prover can have several possible consequences: false proofs, inability to find proofs that should be found, or both. Evidently, problems in the latter two categories will be uncovered more readily than problems that cause only false proofs, but when false proofs do occur, they often introduce noticeable discrepancies between what has been proved and what has not. The output of the theorem prover, such as the proof summary, has to be studied closely enough that it is likely that even flaws that produce only false proofs will eventually be uncovered.

In summary, the operation of the theorem prover is scrutinized more than other parts of the verifier in the course of normal use, and flaws in the theorem prover can be

detected because they result in noticeable deviations from the familiar laws of logic or arithmetic. This alone does not guarantee that all implementation errors will be quickly detected, but it does provide considerable opportunity for concensus through use — opportunity of which verification critics seem to be unaware.

The importance of the simplex algorithm in the theorem prover is a good illustration of the type of reasoning needed in program verification, and why it is sometimes best left to machines. A typical proof of the absence of runtime errors requires reasoning about a set of sparse linear inequalities of program variables. The necessary reasoning could be done by hand, but it tends to be long and uninteresting. Under these conditions, the simplex solver is much faster and more reliable.

Programs based on the simplex algorithm are used heavily by planners, economists, and engineers to make decisions in which there are high penalties for errors. The problems solved in these applications are actually much larger than those which occur in program verification. The standard of reliability in ordinary applications of linear programming is high, but users are given no absolute guarantee of reliability. Is such a standard inadequate for determining the correctness of computer programs?

A verifier in wide use concentrates experience and testing in the same way that compilers and other software tools do. Programs produced using the verifier are used on many cases. If a program has been falsely verified, it is likely that the problem will eventually be discovered by the users of the program, and such a bug will of course be of great interest to the the implementors of the verfier who will then correct the problem. The situation is much the same as the maintanence of compiler implementations. We do not trust compilers to be absolutely correct, but if a compiler has been carefully maintained and in wide use for a long period of time, we have great confidence in it.

Finally, we must mention another factor which contributes to the discovery of bugs in

the verifier: the great challenge[2] of finding a false proof.

[2] and satisfaction

3. Verification and Fault Tolerant Programming

Another canard from verification critics it that program verification encourages the construction of less robust programs. The argument is that if we put trust in proofs of correctness, we will remove safeguards that are needed in case something does go wrong with a program. Thus if a program is falsely verified, the consequences will be more serious than before.

This argument rests on an incomplete view of program verification. To be verifiable, a program must be cleanly designed, and surely that cannot hurt its reliability. Furthermore, verification can contribute to the reliability of error handling in programs. If we have proven that certain errors, such as runtime errors, will not occur, this does not imply that we have to make no provision for these errors, but rather that the value of considering them has been greatly reduced. Depending on the application, we may want to provide protection from hardware errors or illegal data. If we want to have error handling code, program verification gives us the best way to make it reliable. Error handlers are normally one of the least reliable parts of programming because they are executed infrequently and are difficult to test. Through program verification we can consider the effect of error handling systematically, in all the possible cases.

4. Verification and Testing

> Of course we tested it, but why would anyone ever try to set N to -1?
> Programmer's Proverb

Now that we have developed the idea of verification contributing to reliability by helping to form strong agreements about the correctness of programs, we in a position to compare verification with testing. Under what conditions can testing lead to a consensus? Without precise specifications, there can be only weak agreements on the correctness of programs. As we will see, this is not the only similarity between verification and testing if testing is to give strong assurance.

To develop a reliable program with the least effort, it is useful to combine the two methods. Testing is an efficient way to find problems in a new program, but as a program becomes more reliable testing becomes unproductive. When the obvious errors have been corrected after testing, it is time to turn to verification.

Program testing can be many things, ranging from a user selecting a few test values and examining the results to automatic testing based on formal specifications. While one cannot anticipate every conceivable strategy, we do have certain general observations.

Many automated testing strategies attempt to select data very carefully, for instance, to drive a program through a chosen sequence of statements, or to falsify an assertion. But it is exceedingly hard to algorithmically select data which satisfies complicated constraints involving, say, complex data structures or nonlinear arithmetic, and so these strategies are seriously limited.

If many test points are to be used, there must be some automatic way of determining if the program has functioned correctly or not. Thus the problems of formalizing specifications are the same as for program verification. Specification languages for

testing are more restricted because the specifications have to be able to be effectively evaluated, which is not a requirement in program verification.

Testing strategies which treat the program as more than a black box, which do something interesting with the text, must be based on a formal semantics of the programming language. This takes us another step closer to program verification, in that the correctness of the semantic definition and the correctness of its implementation in the tester become issues. The cost of an error here is possible failure to find the errors in an incorrect program, because the tester could decide incorrectly not to test that case of it.

Finally, it is worth noting that if we have formal specifications and language semantics and are trying to decide which parts to test of a program which is already fairly reliable, one of the best ways would be to try to prove the program correct! Parts of a verification which cannot be completed correspond to paths which should be tested.

Conclusion: if automatic testing tools capable to giving strong assurance about program correctness are ever developed, it is likely that they will be based on verification technology such as semantic definitions, specification methods, and theorem provers for reasoning about programs.

5. Shallow verifications vs. Deep proofs

In our view, the natural domain of program verification is in relatively shallow proofs: completely verifying only relatively transparent programs, verifying shallow properties of more complicated programs, verifying deep properties of subtle programs relative to a set of assumed lemmas. There is an important distinction to be made between informal proofs such as those used to justify new algorithms, and verifications. Verification is rigorous analysis of actual programs. The value of verifying relative to a set of assumed lemmas is that one can use a concise kernel of assumptions, developed through careful study, to justify in detail the correctness of complicated programs. One of the basic things that happens in writing a program is that one starts with some known truths and assumptions about data, and gradually diffuses them throughout a long program text until they are no longer readily identifiable.[3] Verification gives assurance that no additional assumptions have been infused.

Full verification of deep properties will probably continue to be too expensive, but there is much value in rigorous checking of even relatively shallow semantic properties. Consider, for instance, the ususal syntax and semantic checking performed by a compiler in a higher level language. Intellectually, nothing could be easier than to write a program that is syntactically correct. Yet the checking performed by compilers is invaluable in actual use. Program verification may be viewed as a means of extending this checking to stronger semantics, freeing the programmer to concentrate fully on the more substantive and creative aspects of a problem, just as current checking in compilers frees the programmer from the necessity of considering certain simple but common kinds of errors.

[3] It is sometimes proposed to make program development a formal activity, so that one would keep track of assumptions throughout the transformation of a program from an initial abstract statement to the final result. This approach may be practical in some cases, but in general it appears to be too rigid. It also seems to be based on the assumption of specifying programs completely, which we feel will be more the exception than the rule.

Profound errors cannot be completely prevented, but many kinds of simple errors can be. An additional benefit of verifying the absence of common errors, as in Runcheck, is that a shallow error often reveals a deeper problem. And so, the process of verifying a program with Runcheck often tells us about much more than its runtime errors.

6. Survey of large programs

At one point during his visit to Stanford University, the author collected large Pascal programs from a number of people at the Artificial Intelligence Laboratory, and spent several days reading them to see what kinds of difficulties would be encountered in proving the absence of runtime errors. Among the programs studied were two Pascal compilers and a hefty micro-assembler. The most important finding was that verifying the absence of runtime errors for these large programs did not appear to involve subtleties of specification or theorem proving beyond what had been encountered with small programs such as those in the Appendix. The problems of proving large programs were much the same as the problems of proving small programs, only spread out over more pages. We are reasonably confident that the approach in Chapter 3, for verifying a moderate sized program, could be applied to nonnumerical programs on the order of, say, 100 pages long, with no more and possibly much less than a proportionate increase in effort.

Of course, as in Chapter 3, we would have to enforce certain restrictions such as absence of aliasing, and we would expect to find a few places where too much detail would be required, so that we would either rewrite a small portion or leave it unchecked. As we have mentioned before, the value of verification in this case is the elimination of surprises from the program. It can still fail for deep reasons, but we can rule out the small slipups which are so common in programming.

Our estimate of the difficulty of verifying a large program is based on the assumption that the individual procedures would all be small; the current implementation of Runcheck cannot efficiently analyze an individual procedure more than about a page long unless the user provides internal assertions to subdivide it logically. It would be very useful to add data flow techniques to Runcheck for fast but undetailed interprocedural analysis. The way we imagine this working is that the entire program would first be subjected to interprocedural data flow analysis, and the

system would make a new program listing containing true assertions discovered thus far. The user would then add additional assertions and work with Runcheck to verify each procedure in detail.

7. Additional techniques for larger programs

As experiments with Runcheck have shown, verification of shallow properties is potentially a highly practical process for real programs (see Chapter 8). This section discusses some of the other ways in which verification can contribute to the reliability of larger programs.

7.1 Core verification

While it is not reasonable to expect a large software system to consist entirely of concise, easily specified algorithms, well structured systems usually contain a core of smaller modules with well defined functions. Outside of the core we would expect to find code which is more diffuse and difficult to specify. But if we can assure that the central parts of a large system function as expected, much will have been accomplished. The cost of program verification in this case is small relative to the size of a system, and because the correct functioning of the core affects everything else, the benefit in reliability is relatively high for each part that is verified.

7.2 Standardization of program specifications

After some years of experience, the writing of certain classes of programs passes from a new experiment to a well understood technique. Similarly, we learn how to specify classes of programs, and develop collections of useful specification concepts. One of the goals of the Stanford program verification project has been the creation of sets of standard specification concepts and lemmas, comparable to standard subroutine libraries. When one is confronted with the problem of verifying a new instance of a familiar type of program, the library specification techniques may not always work as is, but more often, they provide the bulk of the concepts and lemmas needed, and can be readily modified for new applications.

7.3 Program maintenance

It is widely recognized that the major cost in software development is incurred in maintanence over the lifetime of a program and not in the initial design and coding. Syntax based software tools have helped most to reduce the cost of initial coding; they are much less helpful during extended maintainace.

Verification of core components can reduce the cost of maintaince by making a system easier to debug should problems occur. Specifications can be developed incrementally, and modified or extended through experience. In ordinary programming, one can fix a bug, only to have it recur much later, after many other changes have been made. In large systems where complete formal specifications are not used, it might be practical to develop partial specifications during the lifetime of a program. As problems are encountered, instead of merely patching them and hoping that they do not recur, one could specify the absence of the problem and verify its absence in the corrected program. Re-verifying a shallow property of a program after a small modification is generally much less work than the original verification, because most of the specifications and documentation remain unchanged, and the details of the proof will be filled out automatically by the theorem prover.

8. Verification's Impact

At the present time, the direct benefits of program verification are less important than the indirect benefits in the form of increased understanding of programming and programming languages. In recent years, the most significant new ideas in the area of programming languages have not occurred in isolation; without new requirements for programming, research in programming languages would stagnate. Program verification has been one of the most important influences, along with parallelism and distributed systems, artificial intelligence, and more recently, microelectronics.

There is a strong parallel between the directions now being taken in program verification and the path successfully followed several years ago in the field of computer assisted manipulation of mathematical formulas. The MACSYMA [Ma75] project, in particular, has developed a very successful formula manipulation system, but the outcome of work in this field is slightly different from initial expectations. The desire to build powerful formula manipulation systems sparked a fundamental reexamination of certain areas of mathematics, such as the theory of integration, which had been previously felt to be well understood. The results of these investigations included both new understanding of mathematical formulas, and new efficient algorithms for their manipulation. MACSYMA is not a fully automatic system. It is usually used interactively, with the user deciding on what steps to take, and the system then doing lengthly calculations at his command. The system is now widely used by scientific researchers in many fields to do symbolic calculations that would otherwise be intractible [M79].

The parallels between the fields of symbolic manipulation and program verification are striking. Program verification is seeking a more systematic understanding of the basic ideas of programming, ideas which are already as familiar to us as are the basics of doing algebraic manipulations by hand. We have learned that programming need not be entirely haphazard. Valuable new algorithms have been developed for manipulating programs and their proofs.

Also striking are the parallels between certains criticisms of program verification and possible criticisms of automatic formula manipulation which we now know to be unsustainable. First of all, if a computer program for formula manipulation is not absolutely assured to give correct results, how can it possibly contribute to scientific research? In fact, large systems such as MACSYMA do sometimes have bugs, which are detected and corrected in the normal course of use by a community of users. For the applications in which MACSYMA is used, users evidently must feel that the computer is the most cost effective way of doing certain calculations relative to the cost of other methods and the amount of risk of an incorrect answer in the different methods.

Another objection is the high computational complexity of manipulating mathematical formulas, which seems to rule out automatically solving equations in many domains. Formula manipulation systems have found wide use in spite of this, partly because they are interactive and have efficient algorithms for some operations that are not intractible. Apparently the great efficiency of the operations provided more than makes up for whatever costs the user incurs in interacting with a machine and its inflexible formalisms, as against solving a problem completely by hand. Perhaps builders of verification systems should learn from this to provide for a better division of work between user and machine.

For instance, the Stanford Pascal Verifier does not currently permit quantification in assertions. To represent quantified assertions, the user introduces new, uninterpreted predicates, and then adds implicitly quantified axioms in the form of inference rules for use by the theorem prover [Su76, SVG79]. Operation of the verifier is then completely automatic. However, experience has shown that this approach is unworkable in all but the simplest situations. One is forced to constantly balance the rules between generality and efficiency. When the verifier fails to prove a true formula, the user must enter an ordeal of modifying inference rules by excruciating trial and error. How much simpler it would be to permit explicit quantification by requiring the user to supply an instantiation for each instance [We77]!

What should we infer from the great difficulty thus far in applying verification to large programs? Research in program verification has led to successful developments in a number of areas including formal semantics of programming languages, methods for specifying programs, and methods for automating verifications, but the most practical combinations of these techniques will be somewhat different from what was initially envisioned. There are many approaches that are likely to be practical, but we also have to recognize that with some of the most direct applications, the odds are highly unfavorable. We are just beginning to find the areas in which program verification can be applied to greatest advantage.

References

[Ba63] F.L. Bauer, Algorithm 153 Gomory, Comm. ACM 6 (February 1963), 68.

[BM77] J. Bell and M. Machover, *A Course in Mathematical Logic*, North-Holland, Amsterdam, 1977.

[Bo63] J. Boothroyd, Algorithm 201 Shellsort, Comm. ACM 6 (August 1963), p.445.

[BH77] P Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, N.J., 1977.

[Cl79] E.M. Clarke, Programming Language Constructs for Which It is Impossible to Obtain Good Hoare Axiom Systems, J. ACM 26, 1 (January 1979), pp.129-147.

[CH78] P. Cousot and N. Halbwachs, Automatic Discovery of Linear Restraints Among Variables of a Program, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, January 1978.

[DLP79] R.A. De Millo, R.J. Lipton, and A.J. Perlis, Social Processes and Proofs of Theorems and Programs, Comm. ACM 22, 5, May 1979, pp.271-280.

[BLISS] Digital Equipment Corporation, *BLISS-10 Programmer's Reference Manual*, Maynard Massachusetts, 1978.

[En72] H.B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[FO76] L.D. Fosdick and L.J. Osterweil, Data Flow Analysis in Software Reliability, Computing Surveys, Vol. 8, No. 3, September 1976, pp. 305-330.

[GW75] S.M. German and B. Wegbreit, A Synthesizer of Inductive Assertions, IEEE Trans. on Software Engineering, SE-1, 1 (March 1975), pp.68-75.

[Ge78] S.M. German, Automating Proofs of the Absence of Common Runtime Errors, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, January 1978.

[GLO] S.M. German, D.C. Luckham and D.C. Oppen, Extended Pascal Semantics for Proving the Absence of Common Runtime Errors, unpublished manuscript (1977); available from Stanford Program Verification Group.

[Ho69] C.A.R. Hoare, An axiomatic basis for computer programming, Comm. ACM 12, 10 (Oct. 1969), pp.576–581.

[Ho71] C.A.R. Hoare, Proof of a Program: FIND, Comm. ACM 14, 1 (Jan. 1971), pp.39–45.

[HW73] C.A.R. Hoare and N. Wirth, An Axiomatic Definition of the Programming Language Pascal, Acta Informatica, Vol. 2, 1973, pp.335–355.

[Ic79] J.D. Ichbiah et al, Preliminary ADA Reference Manual, in ACM Sigplan Notices, Volume 14, Number 6, June 1979.

[ILL75] S. Igarashi, R.L. London and D.C. Luckham, Automatic Program Verification I: Logical Basis and its Implementation, Acta Informatica, Volume 4, 1975, pp.145–182.

[JW75] K. Jensen and N. Wirth, *Pascal User Manual and Report*, second edition, Springer-Verlag, New York, 1975.

[KL76] R.A. Karp and D.C. Luckham, Verification of Fairness in an Implementation of Monitors, 2nd Intl. Conference on Software Engineering, San Francisco, 1976.

[KR78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, N.J., 1978.

[Kn68] D.E. Knuth, *The Art of Computer Programming, Vol. 1 - Fundamental Algorithms*, Addison Wesley, Reading Mass., 1968.

[Kn71] D.E. Knuth, Mathematical Analysis of Algorithms, in Proceedings of the IFIP Congress 1971, North-Holland Press, 1972.

[La64] H. Langmaack, Algorithm 263 Gomory 1, Collected Algorithms from CACM, (June 1964).

[Lo75] R.L. London, A View of Program Verification, Proceedings of the International Conference on Reliable Software, IEEE, Long Beach, Calif., April 1975, pp.534-545.

[LS77] D.C. Luckham and N. Suzuki, Proof of Termination Within a Weak Logic of Programs, Acta Informatica, Volume 8, 1977, pp.21-36.

[LS79] D.C. Luckham and N. Suzuki, Verification of Array, Record, and Pointer Operations in Pascal, ACM TOPLAS 1, 2 (October 1979), pp.226-244.

[Ma75] Mathlab Group, *MACSYMA Reference Manual*, Version 8, Project MAC, MIT, Cambridge, Ma., 1975.

[M79] Proceedings of the 1979 MACSYMA Users Conference, Washington, D.C., MIT Laboratory for Computer Science, June 1979.

[Se70] J.J. Seppanen, Algorithm 399 Spanning Tree, Collected Algorithms from CACM, (May 1970).

[Si74] R.L. Sites, Proving that Computer Programs Terminate Cleanly, Computer Science Department Report CS 418, Stanford University, May 1974.

[SVG79] Stanford Verification Group, Stanford Pascal Verifier User Manual, Report No. 11, STAN-CS-79-731, Stanford University, March 1979.

[Su76] N. Suzuki, Automatic Verification of Programs with Complex Data Structures, Ph.D. dissertation, Dept. of Computer Science, Stanford University, 1976.

[SI77] N. Suzuki and K. Ishihata, Implementation of an Array Bound Checker, Proc. Fourth ACM Symposium on Principles of Programming Languages, January 1977, pp.132-143.

[We77] B. Wegbreit, Constructive Methods in Program Verification, IEEE Trans. Software Engineering, SE-3, 3 (May 1977), pp.193-209.

[We71] G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinold, New York, 1971.

[Wh56] B.L. Whorf, *Language, Thought, and Reality*, M.I.T. Press, Cambridge, Mass., 1956.

Appendix

This appendix contains lists of programs which have been verified with Runcheck. The examples are divided into three levels of difficulty:

1) examples which can be verified by Runcheck, when the user supplies only the entry and exit assertions.

2) examples which require simple invariants supplied by the user.

3) examples which require more detailed assertions.

The errors checked in most cases are: accessing an uninitialized variable, dereferencing a NIL pointer, subscript or subrange value out of range, and division by zero. Arithmetic overflow was checked in those examples which contain assertions about MAXINT. In part 3, there is a small example of absence of stack overflow for a recursive procedure.

A few examples in parts 1 and 2 cannot be completely verified without a great deal of additional detail. The difficulties are indicated in each case.

Inductive assertions generated automatically by Runcheck are shown in ***Bold Italics***. ***DCOMMENT*** assertions are generated from preliminary analysis of the program text and entry assertions, while underlined INVARIANT assertions are generated from analysis of temporarily unprovable verification conditions.

Appendix - Part 1.

Example 1: Fast linear array search

```
PASCAL
VAR N:INTEGER;
TYPE ARR=ARRAY[1:N] OF INTEGER;

PROCEDURE SEARCH(KEY:INTEGER; A:ARR; VAR I:INTEGER);
GLOBAL (N);
ENTRY DEF(N) ∧ 1≤N ∧ N≤MAXINT;
EXIT 1≤I ∧ I≤N;

BEGIN
  A[N]:=KEY;
  I:=1;
  DCOMMENT 1≤I
  INVARIANT TRUE
  WHILE A[I]≠KEY DO I:=I+1;
END;
```

Example 2: Bubble sort

```
PASCAL
VAR MAXINT:INTEGER;
VAR N:INTEGER;
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

PROCEDURE SORT (VAR A:NARRAY);
GLOBAL(N,MAXINT);
ENTRY DEF(A)∧DEF(N)∧1≤N∧N≤MAXINT;
VAR B:BOOLEAN;
   I, J, TEMP:INTEGER;
BEGIN
I:=1;
J:=1;
DCOMMENT I≤N ∧ 1≤J ∧ 1≤I
INVARIANT TRUE
WHILE (I≤N-1) DO
 BEGIN
    J'←J;
    DCOMMENT J'≤J
    INVARIANT TRUE
    WHILE (J≤N-I) DO
     BEGIN
     IF A[J]>A[J+1] THEN BEGIN TEMP:=A[J]; A[J]:=A[J+1]; A[J+1]:=TEMP END;
     J:=J+1
     END;
    I:=I+1;
    J:=1
 END
END;
```

Example 3: Merging two arrays

```
PASCAL
TYPE INARR=ARRAY[1:100] OF INTEGER;
TYPE OUTARR=ARRAY[1:200] OF INTEGER;

VAR I,J,N:INTEGER;
VAR A,B:INARR; C:OUTARR;

ENTRY DEF(A)∧DEF(B);
EXIT DEF(C);

BEGIN
N:=100;
I:=1;
J:=1;
DCOMMENT 1≤J ∧ 1≤I ∧ I≤N+1 ∧ J≤N+1 ∧ DEFRANGE(1, I+J-2, C)
INVARIANT TRUE
WHILE (I≤N) AND (J≤N) DO
      BEGIN
      IF A[I]≤B[J] THEN BEGIN C[I+J-1]:=A[I]; I:=I+1 END
            ELSE BEGIN C[I+J-1]:=B[J]; J:=J+1 END;
      END;

I'←I;
DCOMMENT I'≤I ∧ I≤N+1 ∧ DEFRANGE(I'+N, I+N-1, C)
INVARIANT TRUE
WHILE I≤N DO BEGIN C[I+N]:=A[I]; I:=I+1 END;

J'←J;
DCOMMENT J'≤J ∧ J≤N+1 ∧ DEFRANGE(J'+N, J+N-1, C)
INVARIANT TRUE
WHILE J≤N DO BEGIN C[J+N]:=B[J]; J:=J+1 END;
END
```

Example 4: Insertion sort

```
PASCAL
VAR N:INTEGER;
TYPE ARR=ARRAY[1:N] OF INTEGER;

PROCEDURE INSERTSORT(VAR K:ARR);
GLOBAL(N);
ENTRY DEF(K)∧DEF(N)∧2≤N;
EXIT TRUE;

LABEL 5;
VAR I,J,X:INTEGER;
BEGIN
J:=2;
DCOMMENT 2≤J ∧ J-N≤1
INVARIANT TRUE
WHILE J≤N DO
      BEGIN
      I:=J-1;
      X:=K[J];
      I'←I;
      DCOMMENT I≤I'
      INVARIANT TRUE
      WHILE X<K[I] DO
            BEGIN K[I+1]:=K[I]; I:=I-1; IF I<1 THEN GO TO 5; END;
 5:   K[I+1]:=X;
      J:=J+1;
      END;
END;
```

Example 5: Sort by selecting the smallest

```
PASCAL
VAR N: INTEGER;
TYPE SARRAY=ARRAY [1:N] OF INTEGER;

PROCEDURE SELECTSORT(A:SARRAY);
GLOBAL(N);
ENTRY N≥1∧DEF(N);
EXIT TRUE;
VAR I, J, K, X: INTEGER;

BEGIN
I:=1;
DCOMMENT I-N≤1 ∧ 1≤I
INVARIANT TRUE
WHILE I<N DO
      BEGIN
      J:=I+1;
      X:=A[I];
      K:=I;
      X'←X; K'←K; J'←J;
      DCOMMENT J-N≤1 ∧ X≤X' ∧ K'≤K ∧ J'≤J
      INVARIANT TRUE ∧ (J>N ∨ A[J]≥X) ⊃ K≤N
      WHILE J≤N DO
            BEGIN
            IF X>A[J] THEN BEGIN X:=A[J]; K:=J; END;
            J:=J+1
            END;
      A[K]:=A[I];
      A[I]:=X;
      I:=I+1;
      END;
END;
```

Example 6: Reading a file into an array, without duplications

This simple program reads integer values from an external file F and stores them without duplication in an array A. After each value is read, the inner loop compares it to the values previously stored in A. In adition to the ususal index bound checks, it is necessary to show that the inner loop accesses only the initialized portion of A

```
PASCAL
VAR N:INTEGER;
      ARR=ARRAY [1:N] OF INTEGER;
      INFILE=FILE OF INTEGER;

VAR A:ARR;  F:INFILE; J,K:INTEGER;
ENTRY DEF(N) ∧ 1≤N;
EXIT DEF(A);

BEGIN
J:=0;
DCOMMENT 0≤J ∧ J≤N ∧ DEFRANGE(1, J, A)
INVARIANT TRUE
WHILE J≠N DO BEGIN
            K:=J+1;
            READ(F,A[K]);
            J'←J;
            DCOMMENT 0≤J ∧ J≤J'
            INVARIANT TRUE
            WHILE J≠0 DO IF A[J]=A[K] THEN
                                  BEGIN
                                    J:=K-1;
                                    GOTO 1;
                                  END
                               ELSE J:=J-1;
            J:=K;
      1:    END;
END
```

Note that on each iteration of the outer loop, J is either unchanged or set to J+1. Because the READ statement is executed on each iteration, the documenter can assert that the array A is initialized in the range 1 to J. This assertion is available on the inner loop to show that only the initialized portions of A are examined.

Example 7: Quicksort

```
PASCAL
TYPE RARRAY=ARRAY [1:100] OF REAL;

PROCEDURE QUICKSORT(VAR A:RARRAY;
                    L,R:INTEGER);
ENTRY DEF(A) ∧(L<R ⊃ (1≤L∧L≤100 ∧ 1≤R∧R≤100));
EXIT TRUE;
VAR X:REAL; VAR LEFT,RIGHT:INTEGER;
BEGIN
IF L<R THEN
      BEGIN
      LEFT:=L; RIGHT:=R; X:=A[L];
      INVARIANT TRUE ∧ RIGHT≤LEFT ⊃ LEFT≤100
      DCOMMENT L≤LEFT∧RIGHT≤R
      WHILE (LEFT<RIGHT) DO
            BEGIN

            RIGHT'←RIGHT;
            INVARIANT TRUE
            DCOMMENT RIGHT≤RIGHT'∧LEFT≤RIGHT
            WHILE (A[RIGHT]≥X) AND (LEFT<RIGHT)
                  DO RIGHT:=RIGHT-1;

            A[LEFT]:=A[RIGHT];

            LEFT'←LEFT;
            INVARIANT TRUE
            DCOMMENT LEFT'≤LEFT∧LEFT≤RIGHT
            WHILE (A[LEFT]≤X) AND (LEFT<RIGHT)
                  DO LEFT:=LEFT+1;

            A[RIGHT]:=A[LEFT]
            END;
      A[LEFT]:=X;
      QUICKSORT(A,L,LEFT-1);
      QUICKSORT(A,LEFT+1,R);
      END;
END.
```

Example 8: Shellsort [Bo63]

```
VAR N,MAXINT:INTEGER;
TYPE ARR=ARRAY[1:N] OF INTEGER;

PROCEDURE SHELL(VAR A:ARR);
ENTRY DEF(A)∧DEF(N)∧1≤N∧MAXINT≥N+1;
EXIT TRUE;

LABEL 1;
VAR I,J,K,M,W:INTEGER;

BEGIN
M:=N DIV 2;
INDEX1←0; M'←M;
INVARIANT TRUE
DCOMMENT  M=M' DIV EXP(2,INDEX1) ∧ 0≤INDEX1
WHILE M≠0 DO BEGIN
K:=N-M;
FOR J:=1 TO K INVARIANT TRUE
          DCOMMENT  1≤J ∧ J≤K+1
     DO BEGIN
     I:=J;
     INDEX2←0; I'←I;
     INVARIANT TRUE
     DCOMMENT  I-I'=-INDEX2*M ∧ 0≤INDEX2
     WHILE I≥1 DO
          BEGIN
          IF A[I+M]≥A[I] THEN GO TO 1;
          W:=A[I]; A[I]:=A[I+M]; A[I+M]:=W;
          I:=I-M
          INDEX2←INDEX2+1;
          END;
   1: END;
M:=M DIV 2;
INDEX1←INDEX1+1;
END;
END;
```

Example 9: Binary Search

```
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

PROCEDURE BINSRCH(A:NARRAY;M,X:INTEGER;
                  VAR Y:INTEGER);
ENTRY  DEF(N)∧1<N∧MAXINT≥2*N+1;
EXIT DEF(Y);

VAR LOW,HIGH,MID:INTEGER;
BEGIN
LOW:=1; HIGH:=N;
INVARIANT TRUE ∧ (LOW≥HIGH ⊃ LOW≤N)
DCOMMENT 1≤LOW ∧ HIGH≤N
WHILE LOW<HIGH DO
  BEGIN
    MID:=(LOW + HIGH)DIV 2;
    IF X<A[MID] THEN LOW:= MID+1 ELSE HIGH:=MID
  END;
IF X=A[LOW] THEN Y:=LOW ELSE Y:=0
END;
```

Example 10: INSITU Permutation [Kn71]

```
PASCAL
VAR N:INTEGER;
TYPE SUBRANGE= 1:N;
TYPE NARRAY=ARRAY [SUBRANGE] OF INTEGER;

FUNCTION P(J:SUBRANGE):SUBRANGE;
ENTRY TRUE;
EXIT TRUE;
EXTERNAL;

PROCEDURE INSITU(VAR X:NARRAY);
GLOBAL(N);
ENTRY (N≥1)∧DEF(N)∧DEF(X);
EXIT  TRUE;

VAR J, K, L, Y: INTEGER;
BEGIN
  J:=1;
  DCOMMENT 1≤J ∧ J-N≤1
  INVARIANT TRUE
  WHILE J≤N DO
  BEGIN
    K:=P(J);
    INVARIANT TRUE ∧ (J<K ⊃ K≤N)
    WHILE K > J DO
      K:=P(K);
    IF K = J THEN
    BEGIN
      Y:=X[J];
      L:=P(K);
      INVARIANT TRUE ∧ 1≤K ∧ K≤N ∧ (L≠J ⊃ 1≤L ∧ L≤N)
      WHILE L ≠ J DO
      BEGIN
        X[K]:=X[L];
        K:=L;
        L:=P(K)
      END;
      X[K]:=Y;
    END;
  J:=J+1;
  END;
END;
```

Example 11: TREESORT

```
PASCAL
VAR ARRAYSIZE:INTEGER;
TYPE TREEARRAY=ARRAY [1:ARRAYSIZE] OF INTEGER;
PROCEDURE TREESORT3(VAR A:TREEARRAY; L:INTEGER);
GLOBAL (ARRAYSIZE);
ENTRY DEF(A)∧2≤L∧L≤ARRAYSIZE;
EXIT TRUE;

VAR WORK:INTEGER; I:INTEGER;

     PROCEDURE SIFTUP(VAR M:TREEARRAY; IO,N:INTEGER);
     GLOBAL (ARRAYSIZE);
     ENTRY DEF(M) ∧ 1≤IO∧IO≤ARRAYSIZE ∧ 1≤N∧N≤ARRAYSIZE ∧ IO≤N;
     EXIT  TRUE;
     LABEL 7;

     VAR COPY,I:INTEGER; J:INTEGER;
     BEGIN
          I:=IO;  COPY:=M[I]; J:=2*I;
          J'←J;
          DCOMMENT IO≤I ∧ I≤J+1 ∧ J'≤J ∧ J≤2*N+2
          INVARIANT TRUE ∧ I≤ARRAYSIZE
          WHILE J≤N DO
               BEGIN
               IF J<N THEN IF M[J+1]>M[J] THEN J:=J+1;
               IF M[J]>COPY THEN
                    BEGIN
                    M[I]:=M[J];
                    I:=J;
                    J:=2*I;
                    END
                   ELSE GO TO 7;
               END;
        7:  M[I]:=COPY;
     END;
BEGIN
     I:=L DIV 2;
     I'←I;
     DCOMMENT 1≤I ∧ I≤I'
     INVARIANT TRUE
     WHILE I≥2 DO
          BEGIN SIFTUP(A,I,L); I:=I-1 END;
     I:=L;
     I''←I;
     DCOMMENT 1≤I ∧ I≤I''
     INVARIANT TRUE
```

```
        WHILE I≥2 DO
            BEGIN
            SIFTUP(A,1,I);
            WORK:=A[1]; A[1]:=A[I]; A[I]:=WORK;
            I:=I-1
            END
    END;
```

Example 12: Gomory all-integer programming [Ba68]

```
PASCAL
VAR M,N:INTEGER;
TYPE TARRAY=ARRAY[1:N-1] OF INTEGER;
TYPE CARRAY=ARRAY[1:N] OF INTEGER;
TYPE MATRIX=ARRAY[0:M, 1:N] OF INTEGER;

FUNCTION ABS(A:INTEGER):REAL; EXIT TRUE; EXTERNAL;
FUNCTION EDIV(A,B:INTEGER):INTEGER; EXIT TRUE; EXTERNAL;

PROCEDURE GOMORY(VAR A:MATRIX);
GLOBAL(M,N);
ENTRY DEF(A)∧DEF(M)∧DEF(N)∧N≥3∧M≥1;
EXIT TRUE;

VAR I,K,J,L,R:INTEGER;
VAR LAMBDA:REAL;
VAR T:TARRAY; C:CARRAY;

BEGIN
  INVARIANT TRUE
  WHILE TRUE DO
     BEGIN
          FOR I:=1 TO M INVARIANT TRUE
                   DO IF A[I,N]<0 THEN BEGIN R:=I; GO TO 2; END;
          GO TO 5;
        2: FOR K:=1 TO N-1 INVARIANT TRUE DO IF A[R,K]<0 THEN GO TO 4;
          GO TO 6;
        4: L:=K;
          L'←L;
          FOR J:=K+1 TO N-1
                   DCOMMENT L'≤L
                   INVARIANT TRUE ∧ L≤N
                   DO IF A[R,J]<0 THEN
             BEGIN
               I:=0;
               INVARIANT TRUE
        1000:     WHILE A[I,J]=A[I,L] DO I:=I+1;
               IF A[I,J]<A[I,L] THEN L:=J;
             END;
          FOR J:=1 TO N-1 DCOMMENT DEFRANGE(1,J-1,T) DO IF A[R,J]<0 THEN
             BEGIN
               IF A[0,L]≠0 THEN T[J]:=EDIV(A[0,J],A[0,L]) ELSE T[J]:=1;
             END;
          LAMBDA:=ABS(EDIV(A[R,1],T[1]));
          FOR J:=2 TO N-1 INVARIANT TRUE DO IF A[R,J]<0 THEN
             BEGIN
```

```
            IF ABS(EDIV(A[R,J],T[J]))>LAMBDA THEN
                    LAMBDA:=ABS(EDIV(A[R,J],T[J]));
          END;
        FOR J:=1 TO N INVARIANT TRUE DO IF J≠L THEN
          BEGIN
            C[J]:=EDIV(A[R,J],LAMBDA);
            IF C[J]≠0 THEN
                FOR I:=0 TO M INVARIANT TRUE
                    DO A[I,J]:=A[I,J]+C[J]*A[I,L];
          END;
    END;
6: %go here if no solution%
5: END;
```

Note: checking the subscripting for the WHILE loop at label 1000 is very difficult. This loop scans down two columns of the matrix A until it finds an index I such that A[I,J] ≠ A[I,L]. The J and L columns always differ in at least one place because the initial value of A contains a diagonal portion, and each column is only changed by adding multiples of another column. While these facts could be formalized in the verifier, it would not be of practical value. The loop could be changed to a FOR loop, or left alone.

Appendix - Part 2.

Example 13: Spanning tree [Se70]

In this example, declarations of the three array types have been made more restricted than would otherwise be necessary, to help express loop invariants. An IF statement at label 2, which terminates the program, is optional but its inclusion simplifies the verification and makes the program more efficient. Without the IF statement, proof of correct subscripting on the array T[1 .. V-1] would involve the fact that a spanning tree for a graph with V vertices has V-1 edges.

```
VAR E,V:INTEGER;

TYPE EARRAY=ARRAY [1:E] OF 1:V;
TYPE VINTARRAY=ARRAY[1:V-1] OF INTEGER;
TYPE VARRAY=ARRAY [1:V] OF 0:E;

PROCEDURE SPANNING(IA,JA:EARRAY; VAR P:INTEGER; VAR T:VINTARRAY);
GLOBAL (E,V);
ENTRY DEF(E) ∧ DEF(V) ∧ 1≤E ∧ 2≤V;
EXIT TRUE;
LABEL 1,2;
VAR I,J,K,C,N,R:INTEGER;
VAR VA:VARRAY;

BEGIN
C:=0;
N:=0;
DCOMMENT 1≤K ∧ K≤V+1 ∧ DEFRANGE(1,K-1,VA)
FOR K:=1 TO V INVARIANT TRUE DO VA[K]:=0;
DCOMMENT 1≤K ∧ K≤E+1 ∧ 0≤N ∧ 0≤C ∧ N≤K-1 ∧ C≤K-1
FOR K:=1 TO E INVARIANT TRUE ∧ (K≠V+N-1 ⊃ K≤V+N-1) DO
BEGIN
2:    IF K-N=V-1 THEN GOTO 1;
      I:=IA[K]; J:=JA[K];
      IF VA[I]=0 THEN
      BEGIN
            T[K-N]:=K;
            IF VA[J]=0 THEN BEGIN
                        C:=C+1;
                        VA[J]:=C;
                        VA[I]:=C;
                        END
                        ELSE VA[I]:=VA[J];
      END
```

```
        ELSE IF VA[J]=0 THEN
        BEGIN
            T[K-N]:=K; VA[J]:=VA[I];
        END
        ELSE IF VA[I]≠VA[J] THEN
        BEGIN
            T[K-N]:=K; I:=VA[I]; J:=VA[J];
            DCOMMENT 1≤R ∧ R≤V+1
            FOR R:=1 TO V INVARIANT TRUE DO
                IF VA[R]=J THEN VA[R]:=I;
        END
        ELSE N:=N+1
    END;
    1: P:=V-E+N;
    END;
```

Example 14: Routines to read in and multiply matrices - check for Overflow.

SUM(A,B,I,J,K) stands for finite sum of A[I][L]*B[L][J] for L from 1 to K.

PC(A,B,MAXINT) is the weakest precondition to multiply A and B with this program without overflow.

PC(A,B,MAXINT) implies that ∀I,J,K, A[I][K]*B[K][J] is inrange, and also ∀I,J,K SUM(A,B,I,J,K) is inrange.

PASCAL

```
VAR M,N,P:INTEGER;
TYPE NVEC=ARRAY[1:N] OF INTEGER;
TYPE PVEC=ARRAY[1:P] OF INTEGER;
TYPE MPARRAY=ARRAY[1:M] OF PVEC;
TYPE PNARRAY=ARRAY[1:P] OF NVEC;
TYPE MNARRAY=ARRAY[1:M] OF NVEC;
TYPE INFILE=FILE OF INTEGER;

VAR MAXINT:INTEGER;
VAR A:MPARRAY;B:PNARRAY;C:MNARRAY;
VAR I,J,K,S:INTEGER;


PROCEDURE READMP(VAR A:MPARRAY);
%initialize A by reading in a matrix.%
GLOBAL(M,N,P,MAXINT);
ENTRY DEF(M)∧DEF(N)∧DEF(P)∧M+1≤MAXINT∧P+1≤MAXINT∧1≤MAXINT;
EXIT DEF(A);
VAR F:INFILE;
VAR I,K:INTEGER;

BEGIN
I:=1;
DCOMMENT 1≤I
INVARIANT DEFRANGE(1,I-1,A)
WHILE I≤M DO BEGIN
      K:=1;
      DCOMMENT 1≤K
      INVARIANT DEFRANGE(1,K-1,A[I])
      WHILE K≤P DO BEGIN READ(F,A[I][K]); K:=K+1 END;
      I:=I+1;
      END;
END;
```

```
PROCEDURE READPN(VAR A:PNARRAY);EXIT DEF(A);EXTERNAL;

PROCEDURE MULTIPLY;
GLOBAL(A,B,C,I,J,K,S,M,N,P,MAXINT);
%main procedure:matrix multiply, c:=a*b%
ENTRY 1≤MAXINT∧DEF(M)∧DEF(N)∧DEF(P)
      ∧ 1≤M ∧ M+1≤MAXINT ∧ 1≤N ∧ N+1≤MAXINT ∧ 1≤P ∧ P+1≤MAXINT;
EXIT DEF(C);
BEGIN
READMP(A);READPN(B);
ASSUME PC(A,B,MAXINT);
I:=1;
DCOMMENT 1≤I ∧ I≤M+1
INVARIANT  DEFRANGE(1,I-1,C)
%assert first I-1 columns of c are defined%
WHILE I≤M DO BEGIN
J:=1;
DCOMMENT 1≤J ∧ J≤N+1
INVARIANT  DEFRANGE(1,J-1,C[I])
%assert first I-1 columns and first J-1 rows of column I are defined%
WHILE J≤N DO
 BEGIN
 S:=0; K:=1;
DCOMMENT 1≤K ∧ K≤P+1
INVARIANT S=SUM(A,B,I,J,K-1)
%note since C is not accessed, no invariant for C needed%
 WHILE K≤P DO
  BEGIN
  S:=S+A[I][K]*B[K][J];
  K:=K+1
  END;
 C[I][J]:=S;
 J:=J+1
 END;
I:=I+1
END
END;
```

This example illustrates a practical limitation of verifying the absence of certain errors, especially arithmetic overflow: the precondition PC for absence of overflow while multiplying A and B is so detailed, that it would be impractical to try to prove it was satisfied each time MULTIPLY was called. Of course, we could prove PC by showing some stronger and simpler condition on the matrices, but in many applications it would be just as well to leave this as a potential source of overflows, and to provide an error handler.

Example 15: Functions for maintaining Queues for Monitors [KL76]

```
PASCAL
VAR N,PN: INTEGER;
TYPE NINTEGER=0:N;
TYPE PNINTEGER=1:PN;
TYPE NARRAY = ARRAY [1:N] OF NINTEGER;
TYPE MONITOR = RECORD LINK: NINTEGER;
                INUSE: INTEGER END;
TYPE PROCARRAY = ARRAY [1:PN] OF MONITOR;

PROCEDURE ADD(M:PNINTEGER; VAR PROCAR: PROCARRAY;
    VAR PLINK: NARRAY; P: NINTEGER );

ENTRY (P ≠ 0)∧DEF(PROCAR)∧DEF(PLINK);
EXIT PROCAR[M].LINK≠0;

 % Insert P into the queue pointed to by the monitor M %
VAR X: INTEGER;

BEGIN
IF PROCAR[M].LINK=0 THEN
BEGIN
      PLINK[P] := 0;
      PROCAR[M].LINK := P;
END ELSE
BEGIN
      X := PROCAR[M].LINK;
      INVARIANT TRUE
      WHILE PLINK[X]≠0 DO
            X := PLINK[X];
      PLINK[P] := 0;
      PLINK[X] := P;
END;
END;

PROCEDURE REMOVE(M: PNINTEGER; VAR PROCAR: PROCARRAY;
      VAR PLINK: NARRAY; VAR RESULT: INTEGER);
GLOBAL(N);
INITIAL PROCAR=PROCAR0;
ENTRY DEF(PROCAR)∧DEF(PLINK)∧ ¬(PROCAR[M].LINK = 0);
EXIT (<PROCAR0,[M].LINK,PLINK[PROCAR0[M].LINK]> = PROCAR) ∧
      (PROCAR0[M].LINK = RESULT) ∧
      1≤RESULT∧RESULT≤N;

VAR X: INTEGER;
BEGIN
% Remove first item from a queue; update distance from head
```

```
   for remaining items %
X := PROCAR[M].LINK;
RESULT := PROCAR[M].LINK;
PROCAR[M].LINK := PLINK[PROCAR[M].LINK];
END;

PROCEDURE ENTER(M, READYQ: PNINTEGER; VAR PROCAR: PROCARRAY;
     VAR PLINK: NARRAY; VAR AP: NINTEGER);
GLOBAL(N);
ENTRY DEF(PROCAR)∧DEF(PLINK)∧DEF(AP)∧ ¬(AP = 0) ;
EXIT TRUE;
VAR TEMPLINK: INTEGER;
BEGIN
% ENTER MUTUAL EXCLUSION STATE   %
IF (PROCAR[M].INUSE = 0) THEN PROCAR[M].INUSE := 1  %
ELSE
BEGIN
   ADD(M,PROCAR,PLINK,AP);
   %  BLOCK(AP);
    NOTE: The procedure ADD by making PCOUNT[AP]
    nonzero (which it does by inserting it into some
    queue), indicates the process AP is blocked
    (inactive or asleep).        %
   IF (PROCAR[READYQ].LINK ≠ 0) THEN
     REMOVE(READYQ,PROCAR,PLINK,TEMPLINK);
   % Removing from the READYQ (if it is not empty) is how
     a process finally gets going.  Of course, in a real
     machine this item would get put into a processor and
     resume execution in that processor.      %
END;
% EXIT MUTUAL EXCLUSION STATE    %
END;


PROCEDURE EXIT(M, READYQ: PNINTEGER; VAR PROCAR: PROCARRAY;
  VAR PLINK: NARRAY);
GLOBAL(N);
ENTRY DEF(PROCAR)∧DEF(PLINK);
EXIT  TRUE;

VAR TEMPLINK: INTEGER;
BEGIN
% ENTER MUTUAL EXCLUSION STATE ;   %
IF PROCAR[M].LINK = 0 THEN PROCAR[M].INUSE := 0
ELSE
BEGIN
   REMOVE(M,PROCAR,PLINK,TEMPLINK);
   ADD(READYQ,PROCAR,PLINK,TEMPLINK);
```

```
    % Adding to the READYQ is how a process is made READY %
% Here, the original algorithm put the calling procedure
  into the READYQ and then removed the head of the READYQ.
  It is more consistend with usage in the rest of these
  routines to delete these two calls, and just let the procedure
  doing the exit resume execution.  %
END;
% EXIT MUTUAL EXCLUSION STATE ;     %
END;

PROCEDURE WAIT(CV, M, READYQ: PNINTEGER; VAR PROCAR: PROCARRAY;
   VAR PLINK: NARRAY; AP: NINTEGER);

GLOBAL(N);
ENTRY DEF(PROCAR)∧DEF(PLINK)∧DEF(AP)∧  ¬(AP = 0);
EXIT TRUE;

%A process AP wishes to wait for condition CV (others
  who request to wait will be served earlier). While
  waiting, wake up the first thing in monitor M if anything
  is there %
VAR TEMPLINK: INTEGER;
BEGIN
% ENTER MUTUAL EXCLUSION STATE; %
IF (PROCAR[M].LINK = 0) THEN
  PROCAR[M].INUSE := 0 ELSE
BEGIN
    REMOVE(M,PROCAR,PLINK,TEMPLINK);
    ADD(READYQ,PROCAR,PLINK,TEMPLINK);
    % Adding to the READYQ is how a process is made READY. %
END;
ADD(CV,PROCAR,PLINK,AP);
% BLOCK(AP);
NOTE: The procedure ADD, by making PCOUNT[AP]
     nonzero (which it does by inserting it into some
     queue), indicates the process AP is blocked
     (nonactive or asleep).       %
IF (PROCAR[READYQ].LINK ≠ 0) THEN
     REMOVE(READYQ,PROCAR,PLINK,TEMPLINK);
% Removing from the READYQ (if it is not empty) is how
     a process finally gets going.  Of course, in a real
     machine this item would get put into a processor and
     resume execution in that processor.      %
% EXIT MUTUAL EXCLUSION STATE; %

END;
PROCEDURE SIGNAL( CV, M, READYQ: PNINTEGER; VAR PROCAR: PROCARRAY;
    VAR PLINK: NARRAY; AP: NINTEGER);
```

```
GLOBAL(N);
ENTRY DEF(PROCAR)∧DEF(PLINK)∧DEF(AP)∧ ¬(AP = 0);
EXIT TRUE;
VAR TEMPLINK: INTEGER;
BEGIN
% ENTER MUTUAL EXCLUSION STATE; %
IF (PROCAR[CV].LINK ≠ 0) THEN
BEGIN
   REMOVE(CV,PROCAR,PLINK,TEMPLINK);
   ADD(M,PROCAR,PLINK,AP);
   % BLOCK(AP);
   NOTE: The procedure ADD, by making PCOUNT[AP]
   nonzero (which it does by inserting it into some
   queue), indicates the process AP is blocked
   (nonactive or asleep).      %
   ADD(¬EADYQ,PROCAR,PLINK,TEMPLINK);
   % Adding to the READYQ is how a process is made READY. %
   REMOVE(READYQ,PROCAR,PLINK,TEMPLINK);
   % Removing from the READYQ (if it is not empty) is how
     a process finally gets going.  Of course, in a real
     machine this item would get put into a processor and
     resume execution in that processor.      %
END;
% EXIT MUTUAL EXCLUSION STATE ;      %
END;
```

Example 16: Deutsch-Schorr-Waite List Marking algorithm [Kn68]

```
PASCAL
LABEL 1,2;
TYPE LIST=↑WORD;
TYPE WORD=RECORD  FL:INTEGER;
                  M:INTEGER;
                  HD:LIST;
                  TL:LIST
        END;
VAR W,Z,Z0,X:LIST;

ENTRY  DEF(Z0)∧DEF(#WORD);
EXIT  TRUE;

BEGIN
      Z:=Z0; X:=NIL;
1:
      ASSERT DEF(X)∧DEF(Z)∧DEF(#WORD);
      IF (Z=NIL) THEN GOTO 2;
      IF (Z↑.M=1) THEN GOTO 2;
      Z↑.M:=1;
      W:=Z↑.HD;
      Z↑.HD:=X;
      X:=Z;
      Z:=W;
      GOTO 1;
2:
      ASSERT DEF(X)∧DEF(Z)∧DEF(#WORD);
      IF X≠NIL THEN
            IF X↑.FL=0 THEN
                  BEGIN
                  X↑.FL:=1;
                  W:=X↑.HD;X↑.HD:=Z;Z:=X↑.TL;X↑.TL:=W;
                  GOTO 1
                  END ELSE
                  BEGIN
                  W:=X↑.TL;X↑.TL:=Z;Z:=X;X:=W;
                  GOTO 2
                  END
END.;
```

Appendix - Part 3.

Example 17: Root and Sentinel (linked list insertion)

```
PASCAL
TYPE REF=↑WORD;
   WORD=RECORD KEY:INTEGER;COUNT:INTEGER;NEXT:REF END;

PROCEDURE SEARCH(X:INTEGER;SENTINEL:REF;VAR ROOT:REF);
GLOBAL (VAR #WORD);
ENTRY (SENTINEL↑.NEXT=NIL) ∧DEF(ROOT)∧DEF(#WORD)
     ∧SENTINEL≠NIL∧ROOT≠NIL
     ∧REACH(#WORD,ROOT,SENTINEL);
EXIT DEF(#WORD);

VAR W1,W2:REF;
BEGIN  W1:=ROOT;
     SENTINEL↑.KEY:=X;
     IF W1=SENTINEL THEN
     BEGIN
          NEW(ROOT);
          ROOT↑.KEY:=X; ROOT↑.COUNT:=1; ROOT↑.NEXT:=SENTINEL;
     END ELSE
     IF W1↑.KEY =X THEN W1↑.COUNT:=W1↑.COUNT+1 ELSE
     BEGIN
          REPEAT W2:=W1; W1:=W2↑.NEXT;
               UNTIL W1↑.KEY=X
               INVARIANT
               (SENTINEL↑.KEY=X)
               ∧W1≠NIL∧W2≠NIL∧SENTINEL≠NIL∧DEF(W2)
               ∧REACH(#WORD,W1,SENTINEL);
          IF W1=SENTINEL THEN
          BEGIN
               W2:=ROOT; NEW(ROOT);
               ROOT↑.KEY:=X; ROOT↑.COUNT:=1; ROOT↑.NEXT:=W2;
          END ELSE
          BEGIN
               W1↑.COUNT:=W1↑.COUNT+1;
               W2↑.NEXT:=W1↑.NEXT;
               W1↑.NEXT:=ROOT; ROOT:=W1
          END
     END;
END;.
```

Example 18: Hoare's FIND [Ho71]

PINVARIANT(I,N,R,A) ≡ ∃p I≤p≤N ∧ R≤A[p]

QINVARIANT(M,J,R,A) ≡ ∃q M≤q≤J ∧ A[q]≤R

```
PASCAL
VAR K: INTEGER;
TYPE SARRAY=ARRAY [1:K] OF INTEGER;
PROCEDURE FIND(F:INTEGER; A:SARRAY);
GLOBAL(K);
ENTRY 1≤F ∧ F≤K ∧ DEF(K);
EXIT TRUE;
LABEL 10;
VAR R,I,J,W,N,M : INTEGER;
BEGIN
M:=1;
N:=K;
DCOMMENT 1≤M ∧ N≤K
INVARIANT  (M≤F)∧(F≤N)
WHILE M < N DO
      BEGIN
      R:=A[F]; I:=M; J:=N;
      I'←I; J'←J;
      DCOMMENT J'≤J' ∧ I'≤I
      INVARIANT  ((I≤J)⊃(PINVARIANT(I,N,R,A)∧QINVARIANT(M,J,R,A)))
      WHILE I≤J DO
            BEGIN
            I''←I;
            DCOMMENT I''≤I
            INVARIANT PINVARIANT(I,N,R,A)
            WHILE A[I]<R DO BEGIN I:=I+1; END;
            J''←J;
            DCOMMENT J≤J''
            INVARIANT QINVARIANT(M,J,R,A)
            WHILE R<A[J] DO BEGIN J:=J-1; END;
            IF I ≤ J THEN
                  BEGIN
                  W:=A[I]; A[I]:=A[J]; A[J]:=W;
                  IF I=J THEN I:=I;
                  I:=I+1; J:=J-1;
                  END;
            END;
      IF F ≤J THEN N:=J ELSE IF I≤F THEN M:=I ELSE GOTO 10
      END;
10:
END;
```

Example 19: Recursive Tree Traversal (absence of stack overflow)

PASCAL

```
TYPE PTR=↑REC;
   REC=RECORD A:PTR; B:PTR END;
VIRTUAL VAR STACKPTR,STACKSIZE:INTEGER;

PROCEDURE WALK(P:PTR);
ENTRY ACYCLIC(P,#REC) ∧ DEF(#REC) ∧ STACKPTR≤STACKSIZE-DEPTH(P,#REC);
EXIT TRUE;

BEGIN
IF P≠NIL THEN BEGIN WALK(P↑.A); WALK(P↑.B) END;
END;
```