ARPA ORDER NO. 2223

ISI/RR-76-46
June 1976

Wm. A. Wulf
Carnegie-Mellon University

Ralph L. London
USC Information Sciences Institute

Mary Shaw
Carnegie-Mellon University

ABSTRACTION and VERIFICATION in ALPHARD: Introduction to Language and Methodology

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) ISI/RR-76-46	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) ABSTRACTION and VERIFICATION in ALPHARD: Introduction to Language and Methodology	5. TYPE OF REPORT & PERIOD COVERED (9) Research Report, 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) William A. Wulf, Carnegie-Mellon University Ralph L. London, ISI Mary Shaw, Carnegie-Mellon University	8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2223
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE (11) (14) June [illegible], 1976 13. NUMBER OF PAGES 49
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 50p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution unlimited.

(15) DAHC15-72-C-0308

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

F44620-73-C-0074

18. SUPPLEMENTARY NOTES

(16) ARPA Order-2223

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

(OVER)

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

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

40752

19. KEY WORDS

abstraction and representation, abstract data types, assertions, correctness, data abstraction, data structures, extensible languages, information hiding, levels of abstraction, modular decomposition, program specifications, program verification, programming languages, programming methodology, proofs of correctness, protection, structured programming, types, verification

20. ABSTRACT

Alphard is a programming language whose goals include supporting both the development of well-structured programs and the formal verification of these programs. This paper attempts to capture the symbiotic influence of these two goals on the design of the language. To that end the language description is interleaved with the presentation of a proof technique and discussion of programming methodology. Examples to illustrate both the language and the verification technique are included.

ABSTRACTION and VERIFICATION in ALPHARD: Introduction to Language and Methodology

Wm. A. Wulf, Carnegie-Mellon University

Ralph L. London, USC Information Sciences Institute

Mary Shaw, Carnegie-Mellon University

June 14, 1976

Abstract: Alphard is a programming language whose goals include supporting both the development of well-structured programs and the formal verification of these programs. This paper attempts to capture the symbiotic influence of these two goals on the design of the language. To that end the language description is interleaved with the presentation of a proof technique and discussion of programming methodology. Examples to illustrate both the language and the verification technique are included.

Keywords and Phrases: abstraction and representation, abstract data types, assertions, correctness, data abstraction, data structures, extensible languages, information hiding, levels of abstraction, modular decomposition, program specifications, program verification, programming languages, programming methodology, proofs of correctness, protection, structured programming, types, verification

The research described here was supported in part by the National Science Foundation (Grant DCR74-04187) and in part by the Defense Advanced Research Projects Agency (Contracts: F44620-73-C-0074, monitored by the Air Force Office of Scientific Research, and DAHC-15-72-C-0308). The views expressed are those of the authors.

Contents

Introduction

The principal subject of this paper is the symbiosis between program verification and programming methodology, especially the way the relationship has affected the design of a particular programming system, Alphard (currently under development at Carnegie-Mellon University). The original design goals for Alphard were concerned with both methodology and verification. We wished to produce a programming environment which supported and encouraged the development of "well structured" programs, and which also made the verification of those programs easier than in existing languages. We have been surprised and extremely pleased at the degree to which these concerns have reinforced each other to produce a coherent system design. Although we shall discuss language design and verification separately, our real goal in this paper is to show that they are not independent, and that when they are treated together a pleasing union results.

Our ultimate concern is with the cost and quality of *real* programs. It is by now generally accepted that programming costs are too high, quality is too low, schedules are too often missed, and so on. We assume that the reader is already familiar with the discussion of the situation and with some of the proposals for remedying it [Baker72, Brooks75, Buxton70, Dahl72, Dijkstra68a, Goldberg73, Gries74, Naur69, Parnas71,72a, Weinberg71, Wirth71, Wulf72].

The area called *programming methodology* or *structured programming* is concerned with those aspects of the current software problem which result from our human limitations in dealing with complexity. Large programs, even not-so-large ones, are among the most complex creations of the human mind. They are often too complex for their creators to understand. This "unmanageable complexity" is at the root of many problems with contemporary software. Structured programming has addressed this situation by attempting to reduce the complexity of programs (or at least their *apparent* complexity), by restricting either the form of the programs (by eliminating the goto, for example[Dijkstra68b]) or the process of creating them (as is the case with *stepwise refinement* [Wirth71]). In both cases the intent is to match the complexity, as we humans perceive it, to the limitations of our understanding.

Problems that arise from repeated modification of large programs are often ignored in the literature on programming methodology. Most large programs are not simply written and run; rather, they are continually modified and enhanced. The same limitations which effectively prevent humans from dealing with the complexity of large programs also prevent them from anticipating all the ways their programs will be used. Thus the initial program is seldom adequate for all its eventual uses, and it experiences constant pressure

for improvement and expansion. Indeed, the more successful a program is, the more likely it is to be modified: only programs no longer in use are safe from this pressure. In many cases the cost of modification exceeds that of initial development, often by a large amount [Goldberg73].

Although modification issues have not received the attention we believe they deserve, the concerns of programming methodology are especially relevant to solving them. Much of the effort involved in modifying an extant program is devoted to simply understanding what is already there. If what's there is overly complex, modifying it can be difficult, time consuming, and susceptible to errors.

Responding to the modification issue adds a dimension to programming methodology. It is no longer adequate for the original programmer to develop the program in a well-structured manner; if the program is to be modifiable, the structure of the development must be retained in the ultimate program text. The future reader must be able to perceive the structure and use it to understand what the program is doing. Thus, a major objective of the Alphard design is precisely retention of this structure.

The research on program verification has been concerned with another approach to alleviating the problems with current software -- proving that the programs we write are in fact implementations consistent with their specifications [Floyd67, Hoare69,72b, London75, Manna74, Naur66]. No matter how clearly we write, we must recognize that programming demands absolute precision. To have real confidence in our programs we must develop them with a degree of precision comparable to that found in mathematics. In short, we must aim toward proofs of our programs, even if the proofs are not in fact carried out.

Program proofs tend to be large (at least as large as the program) and tedious. It is not reasonable to expect them to be done "by hand" as a mathematician would; the human effort would be unreasonable and the probability of error too high. Automatic proof aids will be needed if we are to find proofs with a reasonable amount of effort. Existing automated methods are not strong enough to cope with the complexity of real programs, at least as those programs are currently formulated; this has prohibited routine verification of production programs. The Alphard response, as we shall see, has been an attempt to modularize the proofs so that each individual segment is within the ability of present, or easily attainable, automated proof aids.

Recently, attention has turned to verification of collections of related functions as a means of segmenting the verification task along the same lines as the decomposition of the program itself. For example, proof techniques described by Hoare [Hoare72b] and Spitzen and Wegbreit [Spitzen75, Wegbreit76] can show that a data representation and its associated operations possess the expected properties, provided that the representation is directly manipulated *only* by the associated operations and not by other parts of a program. This decomposition and factorization permit parts of the verification to be

performed for each operator definition instead of for each use. Ultimately, the techniques rely on induction on the number of data operations performed. Related proofs may be found in [Guttag75, 76b, Zilles75].

Well structured, understandable, easily modified, and demonstrably consistent programs can in principle be written in any programming language. In practice, however, we know that the presence or absence of certain features in a language can materially affect all these desirable properties. We also know, from both natural and artificial languages, that the language we use to express our ideas can shape the ideas themselves [Whorf56]. Thus, by choosing language features and structure properly we can hope to exert a positive influence on the programs written in the language.[1]

Instead of starting with an existing language and focusing on either methodology or verification individually, we therefore chose to treat the issues together in a new language design.

This paper, together with its companions [London76, Shaw76b], briefly introduces the Alphard language, discusses the verification issues in this general context, and then elaborates on the language mechanisms suggested by this approach to verification. This cycle is repeated several times for various aspects of language and verification; several examples are developed. The closing section returns to the symbiotic relation between methodological and verification concerns.

Preview of the Alphard Language

A key concept in structured programming is *abstraction*: the retention of the essential properties of an object and the corollary neglect of inessential details. For example, all programming languages provide their users with an *abstract machine* from which inessential details such as the specific assignment of memory locations has been eliminated. Abstraction is important to structured programming precisely because it permits a programmer to ignore inessential detail and thereby reduce the apparent complexity of his task.

Several abstraction techniques have appeared in the literature on structured programming. For example, in *stepwise refinement* or *top-down design*, the top-level, abstract description of a program is refined to a description in a programming language in

[1] Of course, in a certain sense any attempt to design a structured programming language is doomed to failure. A perverse programmer can easily defeat any attempt by the language to *guarantee* clarity or correctness. The language can only encourage good structure and provide the opportunity for verification -- it cannot enforce either one.

a series of progressively more concrete steps [Dijkstra72, Wirth71]. In *modular decomposition* [Parnas72a, 72b], the final (source) version of a program is divided into units; each unit is the realization of some abstraction. Parnas further advocates that the implementation of each of these abstractions be *hidden* from its users lest they inadvertently misuse knowledge of the implementation [Parnas71].

The gross organization of Alphard programs is based strongly on Parnas' ideas, although not on the details of his proposals. This style of program decomposition provides the opportunity to isolate and textually localize all of the details about the implementation of an abstraction. This has several advantages over more traditional organizations:

- The places where modifications must be made are more likely to be close together.

- A smaller portion of the program will have to be reverified when a change is made.

- The user of the abstraction may ignore the details of the implementation.

- It becomes possible to make *absolute* statements about certain things (e.g., data structures) which are independent of even perverse programmers.

- The implementor of the abstraction may (sometimes) ignore the complexity of the environment in which the abstraction will be used.

The specific language mechanism used to capture this style of decomposition is derived from Simula *classes* [Dahl72]; a similar adaptation has also recently appeared in CLU [Liskov74,75a], and related features are beginning to appear in other languages (see, for example, [DataConference76]). At this point we shall only introduce the general nature of the construct and the Alphard notation; more details will follow an explanation of the verification issues.

The abstraction mechanism in Alphard is called a <u>form</u>. It permits the programmer to introduce a new abstraction into the program; in most ways the newly introduced abstraction will resemble a new *type* as that term is used in other programming languages.[2]

---

[2] In general, the abstraction introduced by a <u>form</u> need not be a type in the traditional sense. We use the word "type" informally in this paper, however, and the reader will not be misled too badly by thinking in those terms.

Thus, in an Alphard program one might find a definition such as:

```
form complex=
    beginform
    . . .
    endform
```

This definition introduces a new abstract notion, "complex variable". (Here and in the sequel we shall use ellipses, ". . .", to denote text whose details we wish to ignore for the moment.) The form contains all the information relevant to the Implementation of the abstract notion. In this case, for example, we would find both the definition of the data structure to be used in representing a complex variable (e.g., two real variables) and the definition of a set of operations on them (addition, multiplication, assignment, etc.). The form also gives a formal specification of the abstract properties of these complex variables, but the full story of that must wait a bit.

Once such a definition is written, a programmer can write an *abstract* program using the newly defined notion; variables of the new type may be declared, the defined operations may be performed, and so on. For example, one may write:

```
local x,y,z:complex;
. . .
x←x+y*z;
. . .
```

because certain features of the language allow new functions to be associated with the Infix operations.

All of this is, of course, very similar to the notions found in *extensible languages* [Schuman71]. However, the emphasis is considerably different: we are not interested in general syntactic extension. Rather, we are concerned with *encapsulation*, separating the concrete realization (implementation) of an abstraction from its use in an abstract program. Thus, for example, all of the representational information in a form is inaccessible to the abstract program; only those properties defined in the formal specification are accessible.

So much for a preliminary peek into the nature of Alphard. In the following section we describe a technique for verifying the properties of a form. Since so much of the syntax and semantics of Alphard are tuned to this verification technique, we shall explain the technique first, then present the language via an extended example. For now, the important property of the language is its ability to separate the use of an abstraction from the definition of its concrete representation. The verification technique exploits this separation and permits the implementation (the form) to be verified independently of the abstract program in which it is used.

In order to show as clearly as possible the relation between language and verification we have omitted a number of issues from this discussion of Alphard. These include data representation, reference variables, storage allocation, statement and expression syntax, exception handling, input-output, literals, and other things not needed for this exposition. At least for the programs given here, the reader's intuition and good sense should suffice to fill in the gaps.

Verification of Forms

Our overall strategy for verifying Alphard programs parallels the program decomposition implicit in the notion of a form. We shall presume a relatively small main program expressed in terms of operations on abstract objects natural to the problem. This main program is verified by traditional methods (e.g., inductive assertions [Floyd67, Manna74(chapter 3), Naur66]), treating the specifications of the abstract objects and operations as if they were primitive. Then, to justify the use of the specified properties of the abstract objects we verify that the concrete implementation of each abstraction is consistent with its specifications. In general the implementation of an abstraction will be given in terms of further, *lower level*, abstract objects and operations on them. Thus the verification of the algorithms used to implement an abstraction will be similar to the verification of the most abstract (top level) program. An obvious requirement of this approach is that each of the implementations be *correct*, or verified, if the ultimate program is to be verified. Roughly speaking, the verification will show that the specified relations exist between all abstractions and their implementations so that each implementation "behaves like", or models, its abstraction.

The key to the utility of this approach is separating the proof of each program that uses an abstraction from the proof of the implementation of that abstraction. Several advantages accrue from this separation:

- Individual proofs are kept manageably small.

- Program modifications generally imply reverification of only the affected program portion, usually a single form (exceptions occur when the modification affects the specification of the abstraction implemented by the form).

- Although the entire program can be considered correct only when all portions have been verified, it is feasible for certain portions to be unverified during program development. Alternatively, some verified forms may be available from a library while others may have been developed and verified by a subgroup independently; these forms can be used confidently during the development of further programs or forms.

The remainder of this section explicates a proof methodology which permits this separation. It is based on ideas from Hoare's notable paper on correctness of data representations[Hoare72b].

Suppose that we have an abstract type, T, that "y" is an arbitrary object of type T, and that $A_1,...,A_n$ are abstract operations defined on objects of type T. Our first concern will be to define the objects of this type and the operations on them in a manner which permits a *higher level* program to use these objects and be verified easily. This definition consists of three parts: the specifications, which constitute the user's sole source of information about the form, the representation, which describes the representation and related properties of an object of this type, and the implementation, which contains the definitions of the functions that can be applied to an object.

In the specifications, we first define the class of objects belonging to this type by a predicate which, for reasons which become clear later, is called the *abstract invariant*, I_a. Second, since the abstract type, T, may be defined only under certain assumptions about the parameters supplied when it is created, we capture these assumptions by a predicate, β_{req}. Third, we give another predicate β_{init}, which characterizes the initial value given to an abstract object when it is created. Fourth, we define the abstract operations by their *input-output relations*, using pairs of predicates which characterize their effect. We call these β_{pre} and β_{post}; in Hoare's notation [Hoare69] they say:

$$\beta_{pre}(y) \{ A_i \} \beta_{post}(y)$$

characterizing the effect of the operation A_i by asserting that if the predicate β_{pre} holds before the operation is executed, then β_{post} will hold afterwards. A_i is assumed to read or change only y.

Our next concern will be to characterize a concrete implementation of these abstract objects and operations. Suppose that "x" is the concrete representation of an object of type T, and hence, in general, "x" will be a collection, or *record* of concrete variables. Further, suppose that $C_1,...,C_n$ are the concrete operations which purport to be the implementations of the abstract operations $A_1,...,A_n$. The set of concrete objects is also defined by a predicate, which we shall call the *concrete invariant*, I_c. The relation between a concrete object, x, and the abstract object that x represents may be expressed by a *representation function*, rep:

$$\underline{rep}(x)=y$$

Note that the rep function may be many-one; that is, more than one concrete object may represent the same abstract object. Rep must, however, be defined for all x satisfying I_c.

The concrete operations, C_i, must also be characterized in terms of their input-output relations. To avoid confusion in the sequel we shall refer to these predicates as

the input and output conditions, β_{in} and β_{out}, rather than as pre and post conditions. Thus,

$$\beta_{in}(x) \{ C_i \} \beta_{out}(x)$$

We assume that each C_i alters or accesses variables only in x.

Finally, we shall presume a distinguished concrete operation, C_{init}, which is invoked whenever an object is created; this operation is responsible for initializing the concrete representation.

Now, at an intuitive level, we wish to show that the concrete representation and the implementation of the concrete operations are "correct". More specifically, we wish to show that it is safe for the programmer working at the abstract level to prove the correctness of his program using *only* the abstract specifications of the types he uses: I_a, β_{req}, β_{init}, and (for each abstract operation) β_{pre} and β_{post}. In the sequel, we often discuss an arbitrary function whose corresponding abstract and concrete operations are denoted by the symbols A and C, respectively; our remarks are therefore implicitly quantified over the set of such operations.

We have chosen to break the proof of the correctness of the concrete realization into four steps. The first step establishes the validity of the concrete representation. The second establishes that the concrete initialization operation is sufficient to ensure that β_{init} and I_c hold initially, provided β_{req} is satisfied. The third establishes that the code of the concrete operations is in fact characterized by the input-output assertions, β_{in} and β_{out}, and furthermore that I_c is preserved. The last step establishes the relation between the concrete input-output assertions and the abstract pre and post conditions. After describing the proof steps we discuss the relationship between this methodology and Hoare's.

For the Form

1. Validity of the Representation[3]

   $$I_c(x) \supset I_a(rep(x))$$

2. Initialization of an Object

   $$\beta_{req} \{ C_{init} \} \beta_{init}(rep(x)) \wedge I_c(x)$$

---

[3] This condition is actually slightly stronger than necessary since we only need to ensure that those representations *reachable* by a finite sequence of applications of the concrete operations actually represent abstract objects; in practice, however, the stated theorem is not restrictive since I_c can be made stronger if necessary. Note, by the way, that we need not prove the dual ($I_a(y)$ implies the existence of an x such that $y=rep(x) \wedge I_c(x)$) since this is guaranteed for reachable abstract objects by steps 1-4.

For each function

3. Verification of Concrete Operations

$$\beta_{in}(x) \wedge I_c(x) \ \{ C \} \ \beta_{out}(x) \wedge I_c(x)$$

4. Relation Between Concrete and Abstract Specifications

4a. $I_c(x) \wedge \beta_{pre}(rep(x)) \supset \beta_{in}(x)$

4b. $I_c(x) \wedge \beta_{pre}(rep(x')) \wedge \beta_{out}(x) \supset \beta_{post}(rep(x))$

where the primed variable in step 4b represents the value of that variable prior to the execution of the operation.

Note that steps 1 and 4 are theorems to be proved while 2 and 3 are standard verification formulas. Only the last step, 4, should require further explanation. 4a ensures that whenever the abstract operation A could legally be applied in the higher level, abstract program (that is, whenever β_{pre} holds), the input assertion of the concrete operation, β_{in}, will also hold. 4b ensures that if the concrete operation is legally invoked (that is, $I_c(x) \wedge \beta_{pre}(rep(x'))$ holds), then the output assertion of the concrete operation, β_{out}, is strong enough to imply the abstract post-condition, β_{post}. The four steps are sufficient but not necessary for the proof.

Hoare's similar technique for verifying the correctness of the implementation of an abstraction differs from the one described above in two respects. First, his approach does not deal explicitly with the issue of the validity of the representation, or distinguish explicitly between the concrete and abstract invariants. Second, he did not break the proof into several steps; we did so because we felt it would add clarity, would allow easier modifications both of forms and verifications, and would facilitate mechanical verification. In any case, except for step 1, we shall show that the two techniques are equivalent in the sense that from the proofs of one approach, we can derive the proofs required by the other.

Hoare's technique requires our step 2 and, for each function, a combination of steps 3 and 4 which is expressed in our notation as

$$\beta_{pre}(rep(x)) \wedge I_c(x) \ \{ C \} \ \beta_{post}(rep(x)) \wedge I_c(x)$$

To obtain the proofs required by Hoare's approach from our proofs, merge steps 3, 4a, and 4b, using the rule of consequence:[4] The first premise for the application of the consequence rule is

---

[4] The rule of consequence is:

$$\frac{P \supset Q, \ Q \{ S \} R, \ R \supset T}{P \{ S \} T}$$

$$\beta_{pre}(rep(x)) \wedge I_c(x) \supset \beta_{in}(x) \wedge I_c(x) \wedge \beta_{pre}(rep(x))$$

which is step 4a with $I_c(x) \wedge \beta_{pre}(rep(x))$ added to the conclusion. The second premise is

$$\beta_{in}(x) \wedge I_c(x) \wedge \beta_{pre}(rep(x)) \{ C \} I_c(x) \wedge \beta_{out}(x) \wedge \beta_{pre}(rep(x'))$$

which is obtained by the consequence rule using 3, and then noticing that $\beta_{pre}(rep(x'))$ still holds after C since C does not alter x'. The third premise is 4b with the hypothesis $I_c(x)$ added to the conclusion.

Conversely, to obtain our proofs from Hoare's, first note that β_{in} and β_{out} are not included in Hoare's proofs. We are therefore free to choose β_{in} to be $\beta_{pre}(rep(x))$ and β_{out} to be $\beta_{post}(rep(x))$. Step 3 becomes exactly the combined form, and steps 4a and 4b are trivially provable. Thus the two techniques are equivalent.

In some cases it may be appropriate to show the combined form directly for each function. Hoare proves the theorem that if step 2 and the combined form have been shown to hold for the implementation of some abstraction, then a concrete program using this implementation will produce the (representation of the) same result as an abstract program would have.[5] The proof of this theorem uses induction on the number of applications of operations in the abstract program. Our steps 1 and 2 establish the basis step; steps 1, 3, and 4 are used to establish the induction.

One might expect from this description of the methodology that the relationship

$$rep(x1) = rep(x2) \supset A(rep(x1)) = A(rep(x2))$$

would be true for arbitrary abstract functions A. Unfortunately, it is false. For example, let x1 and x2 be equal but not necessarily identical representations of a set S (i.e., x1 and x2 contain exactly the same elements, but in different orders); let the function A select an arbitrary element from S. The post condition for A is just $x \in S$, which does not specify uniquely which element to select.

In the next section we shall return to the description of Alphard and in particular to how the various pieces of information required by the proof technique are supplied in a form. First, however, we must say a few words about the predicate language in which the β's are expressed. The real issue, of course, is the language used for expressing the *abstract* predicates: I_a, β_{init}, β_{pre}, and β_{post}, since the concrete predicates use the same language as the specification of the next lower level abstractions.

There remains some controversy about the best specification techniques [Liskov75b]. We do not wish to enter that debate here; we are content to await the

---

[5] Assuming, of course, that both the abstract and concrete programs terminate.

emergence of one or more appropriate techniques and then adopt them. For the purposes of this paper, however, we must use some scheme from among the existing techniques. As Guttag [Guttag76a] has noted, the *operational specification* technique we are using seems to be more easily used by current programmers, but may have other problems, such as overspecification. Axiomatic techniques (may) avoid these problems at the expense of being less intuitive (at least until one becomes thoroughly familiar with them). We are neither advocating nor rejecting these two techniques here; Alphard should accommodate both, and we have chosen one we are comfortable with.

In this paper, we shall presume the existence of a suitable collection of recognized mathematical entities, such as integers, booleans, sets, sequences, multisets, matrices, and the operations defined on these entities. We assume that they have been defined precisely and that a rich collection of useful theorems has been proved for each.

Our specifications will be stated in terms of these mathematical objects; in effect they will characterize a possible implementation in terms of the abstract mathematical entities. Thus, for example, in the next section we shall define an implementation of a (restricted) stack. The specification will characterize the stack operations in terms of operations on a sequence, with the sequence itself used to capture the state of the stack.

A precise definition of the notion of a *sequence*, adapted from [Hoare72a], has been included as Appendix A. Although the notion is defined formally there, the following brief informal definition is included here to aid the reader in understanding the examples which follow.

$\langle s_1,\ldots,s_k \rangle$	denotes the sequence of elements specified; in particular, "<>" denotes the empty sequence, "nullseq".
$s \sim \langle x \rangle$	is the sequence which results from concatenating element x at the end of sequence s.
length(s)	is the length of the sequence "s".
first(s)	is the first (leftmost) element of the sequence "s".
trailer(s)	is a sequence derived from "s" by deleting the first element.
last(s)	is the last (rightmost) element of the sequence "s".
leader(s)	is a sequence derived from "s" by deleting the last element.
seq(V,n,m)	where "V" is a vector and "n" and "m" are integers, is an abbreviation for the sequence "$\langle V_n,V_{n+1},\ldots,V_m \rangle$"; alternatively, $seq(V,n,m)=seq(V,n,m-1) \sim V_m$.

Note: first, trailer, last, and leader are undefined for "<>".

Introduction to Alphard

This section is an informal discussion of the Alphard language facilities which support the verification technique introduced above. Since we are primarily concerned with structural and verification issues we shall not concern ourselves with minor syntactic aspects of the language or with those (sometimes major) features of the language which do not bear directly on these issues. We expect that the reader's familiarity with other languages will be adequate for him to infer both the syntax and semantics of those constructs whose formal definition is omitted.

Much of the exposition is by example. We develop a definition of stacks and a program which uses stacks. These examples illustrate both the abstract definition facility and the interaction of verification considerations with language. We chose the stack for an example because it is familiar to most readers and because the Alphard program can be compared to other descriptions.

Forms

Imagine that while designing some program we found it desirable to use the notion of a stack -- in particular, a stack whose elements are integers. We presume that our language does not contain stacks as a primitive concept, as indeed Alphard does not, so we want to introduce it as a new abstraction. Suppose further that an a priori depth limit is known or desired, so we need not define a general stack mechanism, only one which behaves like a stack so long as its depth does not exceed some predetermined maximum.

We shall lean heavily on the verification methodology developed above to explain the rationale for the various components of a form definition. We shall present the definition piecemeal, with each piece corresponding to some aspect of the verification technique. Starting at the top, the abstraction of a finite-depth stack of integers will be defined by a form such as:

```
form istack(n:integer)=
    beginform
    . . .
    endform;
```

where "n" is the maximum permissible depth of the stack. Note that we must carefully distinguish between the abstract concept introduced by such a definition and an ***instance***

of that concept. In general there may be many instances of an abstraction. Instances of abstractions are introduced into an Alphard program in several ways, but a common one is by *declarations*. Thus,

```
local x:istack(50);
```

has the effect of creating an instance of an istack and giving the name "x" to this particular instantiation. In the jargon of programming languages, this declaration *binds* the name "x" to an instantiation of istack.

We must now decide what the abstract properties of our stack are to be. We must decide both what operations the abstract program shall be allowed to perform and what effects these operations shall have. In this case we shall allow only four operations: "push" makes a new entry at the top of the stack, "pop" deletes the current top element of the stack, "top" returns the value of the current top element of the stack, and "empty" returns "true" iff the stack is empty. (Obviously we could have chosen a more comprehensive set, but this will suffice for our first example.)

The abstract program which uses the notion of an istack will apply these operations to instances of the abstraction. The form must provide a precise definition of these operations together with the concrete representation and operations to be used in implementing them. Thus, in general, a form is composed of three parts: specifications, representation, and implementation.

```
form istack(n: integer) =
    beginform
    specifications . . .;
    representation . . .;
    implementation . . .;
    endform;
```

At the very least the specifications must provide the names of the operations supplied by the form together with the types of their arguments and results. In order for the user to be able to understand and use the abstraction solely in terms of the specification, and to permit verification, we must also include (1) a definition of the abstract domain, (2) the initial value of each entity of the abstract type, and (3) the pre and post conditions for each operation. Using the mathematical notion of a sequence, defined earlier, we can write:

```
form istack(n: integer) =
    beginform
    specifications
        requires n>0;
        let istack = < . . . x_i . . . > where x_i is integer;
        invariant 0≤length(istack)≤n;
        initially istack=nullseq;
        function
            push(s:istack, x:integer) pre 0 ≤ length(s) < n post s=s'~x,
            pop(s: istack) pre 0 < length(s) ≤ n post s = leader(s'),
            top(s: istack) returns x: integer
                    pre 0 < length(s) ≤ n post x = last(s'),
            empty(s: istack) returns b: boolean
                    post b = (s=nullseq);
    representation . . .;
    implementation . . .;
    endform;
```

Note how various pieces of information about the abstraction implemented by the form are introduced: the requires clause specifies β_{req}, the invariant clause specifies I_a, the initially clause specifies β_{init}, and each of the function clauses specifies β_{pre} and β_{post} for that function.[6] Furthermore, no particular implementation is demanded or precluded.

In this case, then, the notion of an istack is explicated in terms of the mathematical notion of a sequence of bounded length. The operation "pop", for example, is defined to produce a new sequence which is just like the old one except that its last element has been deleted. (As before, the primed symbols in the post conditions, e.g., s', refer to the value of the (unprimed) symbol prior to execution of the operation.)

This particular example allows us to illustrate something which was awkward to introduce in the more abstract discussion in the previous section. Because the form may be parameterized to allow each user to select his own maximum depth, it is more properly a "type generator" (that is, a definition of a set of types) than a simple type definition. Although we will expand on this point at some length in a subsequent section, we note here that not all values of the parameters may make sense. In this case, for example, a stack of negative size is senseless. Restrictions on the parameters are conveniently expressed in β_{req}, that is, the requires portion of the specifications.

---

[6] To shorten the pre, post, in, and out conditions in this paper, we often, by convention, omit assertions about variables which are completely unchanged. Thus, for example, we have omitted s=s' from the post condition of top. Such omitted assertions are nevertheless used in the proof steps. We also generally avoid in our proofs the legitimate concerns expressed in the term "clean termination" -- such matters as array bounds checks, overflow, division by zero, and other inexecutable operations.

The representation portion defines the data structure which each instantiation of the form will use to represent the abstraction. It also specifies: (1) the initialization to be performed whenever the form is instantiated, (2) the rep function, which relates concrete to abstract descriptions, and (3) the concrete invariant. Thus, this section provides the major information relating an abstract entity and its concrete representation.

For this example we have chosen a simple representation for the stack. A vector holds the contents of the stack and an integer variable points to the top of the stack.

```
form istack (n: integer):
        beginform
        specifications . . .;
        representation
                unique v: vector(integer,1,n), sp: integer init sp ← 0;
                rep (v,sp) = seq(v,1,sp);
                invariant 0 ≤ sp ≤ n;
        implementation . . .;
        endform;
```

The first clause of the representation portion describes the concrete data structure(s) used to represent the abstraction; the key word unique used here indicates that the following data structure(s) are unique to each instantiation as opposed to being shared by, or common to, all instantiations. The rep clause specifies the representation function which maps concrete objects to abstract ones. The invariant clause specifies I_c. Also, note the init clause attached to the data structure declaration; this is the distinguished operation, C_{init}, mentioned in the previous section. The initialization operation is automatically invoked whenever an instantiation of the form is created, and is responsible for establishing β_{init}.

We would also like to point out the use of the names "vector" and "integer" in this example. These are *not* primitive types of the language; they are simply form names. They happen to be the names of forms which will be automatically provided along with the compiler, but they are not special in any other way.

From experience in writing forms, we have found that it is convenient to add another piece of information to the representation: a set of state definitions. These states are merely a shorthand for a set of boolean conditions, but, as we shall see below, they help to accent certain interesting situations. A more complete version of the representation portion of the form is thus:

```
form istack (n: integer):
    beginform
    specifications . . .;
    representation
        unique v: vector(integer,1,n), sp: integer init sp ← 0;
        rep (v,sp) = seq(v,1,sp);
        invariant 0 ≤ sp ≤ n;
        states
            mt when sp = 0,
            normal when 0 < sp < n,
            full when sp = n,
            err otherwise;
    implementation . . .;
    endform;
```

The implementation portion of the form contains the bodies of the functions listed in the specifications, together with their concrete input and output assertions (β_{in} and β_{out}). In defining these function bodies we make use of the states defined in the representation part. The state of the representation is determined when any function in the form is invoked, but is not re-evaluated as changes to the representation are made within a function body. Thus the state may be used, as in this example, to select one of several possible bodies for a function when it is called. In this particular example the ability to select alternate bodies is used only for error detection, but it is certainly not limited to this use.

```
form istack(n: integer) =
    beginform
    specifications . . .;
    representation . . .;
    implementation
        body push out (s.sp = s.sp' + 1 ∧ s.v = ∝(s.v',s.sp,x))=
                mt,normal:: (s.sp ← s.sp + 1; s.v[s.sp] ← x);
                otherwise:: FAIL;

        body pop out (s.sp = s.sp'-1) =
                normal,full:: s.sp ← s.sp-1;
                otherwise:: FAIL;

        body top out (x = s.v[s.sp]) =
                normal,full:: x ← s.v[s.sp];
                otherwise:: FAIL;
```

```
        body empty out (b = (sp=0)) =
                        normal,full:: b ← false;
                        mt:: b ← true;
                        otherwise:: FAIL;

    endform;
```

Since the states are used to select one of several alternative bodies for a function, the state descriptions may be used as additional input assertions for the body selected. Thus, for step 3 of the proof we may add to the precondition the disjunction of the (state) conditions that can cause the selection of that body. The notation "$\alpha(V,i,x)$", which is used in the output assertion of "push", denotes a vector identical to "V" except that $V_i=x$. Finally, the symbol FAIL used above is intended to connote failure; we prefer to avoid a detailed discussion of the exception mechanism in this paper and hence will avoid further elaboration of this symbol here.

Naming and Scope

The previous section dealt with the general organization of forms; in this section we shall deal with some of the linguistic details of naming and scope. There are two issues to be discussed here: one is almost at the level of syntactic detail, but the other is fundamental to the ability of a form to encapsulate an abstraction through *information hiding*. Given the goals of this paper we would normally omit the first of these; they are closely related, however, so we shall discuss them in sequence.

Consider the previous definition of "istack". We said earlier that one or more instantiations of this abstraction can be created by *declarations*, and that the operations defined in the form may then be applied to them. For example,

```
local s1,s2: istack(10);
. . .
push(s1,5);
. . .
if top(s2)=23 then . . .
```

But now suppose that another abstraction, call it "rstack", had been defined in the same program and that it also defined a function "push". We then have to decide which push operation is being invoked in any given situation The answer, of course, is that the interpretation of operation names is context dependent. We know that in the example above the correct "push" is the one in "istack" because its first parameter is an instance of the istack abstraction. The point can be made clearer by a slight change in notation; a construct of the form "name1.name2" is called a *qualified name*, its first component must

be the name of an instance of some abstraction and its second component must be the name of a function defined in the appropriate form. Thus,

```
s1.push(5);
```

is an invocation of the "push" function defined in the form of which "s1" is an instance. Although this notation is more explicit about the operation named, it has an asymmetry which is often displeasing.[7] Thus, Alphard permits both styles of naming, i.e.,

```
f(p1,p2,. . .,pn) ≡ p1.f(p2,. . .,pn)
```

Although this convention also has some problems, they do not arise in the examples in this paper (see [Geschke75, Ross70] for discussions of the "uniform referent" problem); we shall use whichever notation seems most appropriate in a given instance. In all cases, however, functions are defined as though the abstraction instance were its first parameter.

The more substantive issue is that of scope -- which names are defined where. Consider the "istack" form again. Inside the form several names are defined; some of these are the abstract operations, e.g., "push", others are related to the representation, e.g., "sp". From the discussion above we know that the operation names are available outside the form as qualifiers of instance names. In Alphard, however, names such as "sp" are *not* available outside the form.

Only names defined in the specifications part of the form are legal outside the form definition (inside is another matter). If names such as "sp" were legal outside the form, the abstract program could access, and possibly modify, the concrete representation. If this were allowed, both theoretical and practical difficulties would arise. First, we could not partition the proof technique as described above; specifically, we could not ensure that the concrete invariant was preserved between function invocations. Second, since the representational information would no longer be hidden it would no longer be safe to modify a form under the sole restriction that specified properties were preserved. We would instead have to examine all the uses of the abstraction to be sure that the representational information was not being used in some clever, but obscure, way.

In summary, only the names appearing in the specification part of a form are legal qualifiers outside the form definition. In the examples so far all such names have been function names; as we shall see in future examples, this need not always be the case.

---

[7] For example, for binary commutative operations such as "plus" it seems unnatural to write "x.plus(y)" rather than "plus(x,y)".

The general scope rules in Alphard are Algol-like,[8] but with two important exceptions:

1. Only those names appearing in the specification part of a form may be used as qualifiers outside the form definition. (Note: all the names defined in a form may be used as qualifiers inside the same form definition.)

2. Only form names obey the usual block-structure convention on entering a form. Specifically, only those variables defined outside a form which are passed as parameters are accessible inside the form body.

The earlier paragraphs dealt with the rationale for the first of these restrictions. The second restriction is imposed so that there are no free variables in a form body; this ensures that any dependency of the form on its environment is explicated in its parameter list.

An Aside on Primitive Forms

A basic question which must be answered in the design of any language is which primitive types should be provided by the language and which should be left for the user to define. The Alphard position is that *all* types but one are defined by forms and, at least conceptually, could be (re)defined by the user. (The one primitive form which can be specified but not implemented in Alphard corresponds roughly to the untyped memory of conventional computers.) To be usable, however, a collection of familiar and useful forms are defined by a *standard prelude* [vanWijngaarden69, chapter 10], which is automatically inserted at the beginning of every user's program. Throughout this paper we shall use notions such as integer, real, boolean, vector, and so on; the reader may presume that these are either provided by the standard prelude or have been explicitly defined by other forms in the same program. In all cases, however, the reader should assume that these provide the familiar facilities.

Example of a form Verification: Restricted Stacks

In this section we shall illustrate the verification technique on the istack form of the previous section. First, however, let's pull together the pieces of the istack definition:

[8] By *Algol-like* we simply mean that the interpretation of a name depends upon its nearest definition in a potentially nested, static block structure.

```
form istack(n: integer) =
    beginform
    specifications
        requires n>0;
        let istack = < . . . x_i . . . > where x_i is integer;
        invariant 0≤length(istack)≤n;
        initially istack=nullseq;
        function
            push(s:istack, x:integer) pre 0 ≤ length(s) < n post s=s'~x,
            pop(s: istack) pre 0 < length(s) ≤ n post s = leader(s'),
            top(s: istack) returns x: integer
                    pre 0 < length(s) ≤ n post x = last(s'),
            empty(s: istack) returns b: boolean
                    post b = (s=nullseq);

    representation
        unique v: vector(integer,1,n), sp: integer init sp ← 0;
        rep (v,sp) = seq(v,1,sp);
        invariant 0 ≤ sp ≤ n;
        states
            mt when sp = 0,
            normal when 0 < sp < n,
            full when sp = n,
            err otherwise;

    implementation
        body push out (s.sp = s.sp' + 1 ∧ s.v = α(s.v',s.sp,x))=
                    mt,normal:: (s.sp ← s.sp + 1; s.v[s.sp] ← x);
                    otherwise:: FAIL;

        body pop out (s.sp = s.sp'-1) =
                    normal,full:: s.sp ← s.sp-1;
                    otherwise:: FAIL;

        body top out (x = s.v[s.sp]) =
                    normal,full:: x ← s.v[s.sp];
                    otherwise:: FAIL;

        body empty out (b = (sp=0)) =
                    normal,full:: b ← false;
                    mt:: b ← true;
                    otherwise:: FAIL;

    endform;
```

In the verification of istack, which is given next, the precondition for each body is the conjunction of its in clause (which is defaulted to "true") and the union of the state conditions for which that body is selected.

For the form

1. Representation validity
   Show: $0 \le sp \le n \supset 0 \le length(rep(x)) \le n$
   Proof: length(rep(x)) = length(seq(v, 1, sp)) = sp.

2. Initialization
   Show: $n>0$ { sp←0 } rep(v, 0) = nullseq $\wedge$ $0 \le sp \le n$
   Proof: rep(v, 0) = seq(v, 1, 0) = <>, i.e., nullseq

For the function push

3. Concrete operation
   Show: $(0=s.sp \vee 0<s.sp<n) \wedge 0 \le s.sp \le n$ { s.sp←s.sp+1; s.v[s.sp]←x }
   $s.sp=s.sp'+1 \wedge s.v=\alpha(s.v', s.sp, x) \wedge 0 \le s.sp \le n$
   Proof: $0 \le s.sp<n \supset 0 \le s.sp+1 \le n$

4a. β_{in} holds
   β_{in} is true

4b. β_{post} holds
   Show: $0 \le s.sp \le n \wedge 0 \le length(rep(s.v, s.sp')) < n \wedge s.sp=s.sp'+1 \wedge$
   $s.v=\alpha(s.v', s.sp, x) \supset s=s'\sim x$
   Proof: s=rep(s.v, s.sp) = seq(s.v, 1, s.sp'+1) = seq(s.v, 1, s.sp')~s.v[s.sp] =
   seq(s.v', 1, s.sp')~x = s'~x

For the function pop

3. Concrete operation
   Show: $0<s.sp \le n \wedge 0 \le s.sp \le n$ { s.sp←s.sp-1 } $s.sp=s.sp'-1 \wedge 0 \le s.sp \le n$
   Proof: $0<s.sp \le n \supset 0 \le s.sp-1 \le n$

4a. β_{in} holds
   β_{in} is true

4b. β_{post} holds
   Show: $0 \le s.sp \le n \wedge 0<length(rep(s.v, s.sp')) \le n \wedge s.sp=s.sp'-1 \supset s=leader(s')$
   Proof: s=rep(s.v, s.sp) = seq(s.v', 1, s.sp'-1) = leader(s'). Note that
   leader(s') is defined since $s.sp' \ge 1$

For the function top

3. Concrete operation
   Show: $0<s.sp \le n \wedge 0 \le s.sp \le n$ { x←s.v[s.sp] } $x=s.v[s.sp] \wedge 0 \le s.sp \le n$
   Proof: Clear

4a. β_{in} holds
   β_{in} is true

4b. β_{post} holds
Show: 0≤s.sp≤n ∧ 0<length(rep(s.v, s.sp'))≤n ∧ x=s.v[s.sp] ⊃ x=last(s')
Proof: x=s.v[s.sp] = s.v'[s.sp'] = last(s'). Last(s') is defined since s.sp'≥1

For the function empty

3. Concrete operation
(Normal, full) Show: 0<s.sp≤n ∧ 0≤s.sp≤n { b←false } b = (s.sp=0) ∧ 0≤s.sp≤n
Proof: 0<s.sp ⊃ false = (s.sp=0)
(Mt) Show: s.sp=0 ∧ 0≤s.sp≤n { b←true } b = (s.sp=0) ∧ 0≤s.sp≤n
Proof: s.sp=0 ⊃ true = (s.sp=0)

4a. β_{in} holds
β_{in} is true

4b. β_{post} holds
Show: 0≤s.sp≤n ∧ b = (s.sp=0) ⊃ b = (s=nullseq)
Proof: b = (s.sp=0) = (rep(s.v,s.sp)=nullseq) = (s=nullseq)
QED

The condition n≥0 is used implicitly in this proof. The stricter n>0 is needed only to show that the four states are disjoint. Finally, note that the union of the mt, normal, and full states includes I_c and that β_{pre} for each function and I_c specifically exclude the states that would trigger the otherwise alternative for the body. We therefore omit verifications involving FAIL.

Generalizing Form Definitions

The form defines the abstract notion of a stack-of-integers, but what does the fact that the items to be stacked are integers have to do with it? It seems that the *abstract* notion of a stack ought to be independent of the kinds of things being stacked.[9]

We would like to be able to define a form such as

```
form stack(T:form, n:integer)=
  beginform
  . . .
  endform
```

and then create instantiations with statements such as

---

[9] Perhaps one can argue that the fact that all items in a *particular* stack are the same type, e.g., integers, is an abstract property of a stack, but it would be unfortunate if we had to define separate forms for stacks of integers, stacks of reals, stacks of characters, and so on.

```
local si:stack(integer,35), sr:stack(real,14);
```

which would make "si" a stack of integers and "sr" a stack of reals.

We shall do essentially this, but as we introduce this facility we must be very careful to retain the validity of the verification technique. In fact, we want to ensure something stronger: that the resulting proofs are not complicated by the introduction of this additional flexibility. Thus, we shall start with a careful examination of the proof appearing in the preceding section.

Specifically, let's observe how the proof depends upon the fact that the items being stacked are integers. A careful reading of the proof of istack reveals that it depends only upon the property of the items that we have an assignment operation which obeys the *assignment axiom*.[10] The reader is encouraged to examine the proof to verify that this is in fact the only property required, and therefore to see that the proof would be valid for any type of item possessing this assignment axiom.

Returning to the language issues, what we want is a means for stating that the parameter "T" above cannot be just any form name; it must be the name of a form which supplies the properties required by the proof (and, of course, by the bodies of the concrete operations). The general mechanism used to accomplish this will be discussed below; for the moment we will consider only the special case which handles the stack example. With this addition the form "stack" has become a "type generator" as mentioned above rather than a simple type definition.

We shall append a bracketed list $\langle a_1,\ldots,a_n \rangle$ to a formal parameter specification to denote that the properties $a_1,\ldots,a_n$ are *required* of a corresponding actual parameter. Thus, in the present case we may write the stack form header as:

```
form stack(T:form<←>, n:integer)=
  beginform
  . . .
  endform
```

The "<←>" attached to the form parameter asserts that the actual form names used in this position must provide an assignment operation. The specifications part of the actual parameter form must assert the availability of this operation and assure that it obeys the assignment axiom. We shall discuss these issues in greater detail below, but first we shall give the full stack definition and a verification of a program using it. The full stack

---

[10] The assignment axiom is:

$$P_e^x \{ x \leftarrow e \} P$$

if x is a simple variable. For subscripted variables the meaning of x[i] := e is x := α(x,i,e) as in [Hoare72a].

definition differs from the version at the beginning of the previous section only in the nine italicized lines, which are the ones that previously referred to "istack" or "integer". Its proof is identical to that given above.

```
form stack(T:form<↔>, n:integer)=
    beginform
    specifications
        requires n>0;
        let stack = < . . . x_i . . . > where x_i is T;
        invariant 0≤length(stack)≤n;
        initially stack=nullseq;
        function
            push(s:stack, x:T) pre 0 ≤ length(s) < n post s=s'~x,
            pop(s:stack) pre 0 < length(s) ≤ n post s = leader(s'),
            top(s:stack) returns x:T
                    pre 0 < length(s) ≤ n post x = last(s'),
            empty(s: istack) returns b: boolean
                    post b = (s=nullseq);

    representation
        unique v: vector(T,1,n), sp: integer init sp ← 0;
        rep (v,sp) = seq(v,1,sp);
        invariant 0 ≤ sp ≤ n;
        states
            mt when sp = 0,
            normal when 0 < sp < n,
            full when sp = n,
            err otherwise;

    implementation
        body push out (s.sp = s.sp' + 1 ∧ s.v = ∝(s.v',s.sp,x))=
                    mt,normal:: (s.sp ← s.sp + 1; s.v[s.sp] ← x);
                    otherwise:: FAIL;

        body pop out (s.sp = s.sp'-1) =
                    normal,full:: s.sp ← s.sp-1;
                    otherwise:: FAIL;

        body top out (x = s.v[s.sp]) =
                    normal,full:: x ← s.v[s.sp];
                    otherwise:: FAIL;
```

```
        body empty out (b = (sp=0)) =
                normal,full:: b ← false;
                mt:: b ← true;
                otherwise:: FAIL;

    endform;
```

Using Stacks in a Program

Once the stack form is defined, programs may declare and use stacks. The following program uses a stack as defined by this form to traverse a (finite) binary tree and count its tips. It also uses iteration and an explicit stack of binary trees [Burstall74, London75]. A binary tree is defined recursively to be either *nil* or to have a *left son* and a *right son* which are both binary trees. The number of tips is defined recursively by

tips(t) = if t=nil then 1 else tips(leftson(t))+tips(rightson(t))

We shall not define a binary tree form explicitly, but shall presume that it meets at least the specifications

```
isleaf(t:binarytree) returns b:boolean post b = (t=nil),
left(t:binarytree) returns u:binarytree pre t≠nil post u=leftson(t'),
right(t:binarytree) returns u:binarytree pre t≠nil post u=rightson(t')
```

We shall also presume a tree assignment operation satisfying the assignment axiom. In stating the maximum permissible depth of the stack we use the height function defined by

height(t) = if t=nil then 0 else 1+max(height(leftson(t)), height(rightson(t)))

Suppose the tip counter is specified by

function tipcount(t:binarytree) returns count:integer post count=tips(t)

Then the body of the function tipcount might be

```
body tipcount out (count=tips(t)) =
  begin
     unique s:stack(binarytree, max(height(t),1)), x:binarytree;
     x←t; count←1;
     invariant tips(t) = count - 1 + tips(x) + SIGMA_{u∈s}tips(u);
     while ¬ empty(s) ∨ ¬ isleaf(x) do
        if isleaf(x) then (count←count+1; x←top(s); pop(s))
                     else (push(s, right(x)); x←left(x));
  end
```

Throughout the body of tipcount the stack s means the abstract definition in terms of a sequence. In particular, $SIGMA_{u\in s}f(u)$ means 0 if s=nullseq and otherwise

$$f(last(s)) + SIGMA_{u\in leader(s)}f(u)$$

We shall verify the concrete operation of this body (i.e. proof step 3). Note first that the requires clause (n>0) of the stack form is satisfied. We shall use the usual proof rule for the while statement.[11] Four verification conditions suffice; they are in the form obtained by backward substitution with each function operation of a form replaced by its post condition.

1. (entry to while)
   Show: $tips(t) = 1 - 1 + tips(t) + SIGMA_{u\in nullseq}tips(u)$
   where "nullseq" is obtained from the initially clause of stack.
   Proof: The SIGMA term is 0.

2. (while to exit)
   Show: $tips(t) = count - 1 + tips(x) + SIGMA_{u\in s}tips(u) \wedge \neg (s\neq nullseq \vee x\neq nil) \supset count = tips(t)$
   Proof: The SIGMA term is 0 because s=nullseq. tips(x)=1 since x=nil.

---

[11] The while rule is:

$$\frac{P \wedge B \{ S \} P}{P \{ \text{while } B \text{ do } S \} P \wedge \neg B}$$

This is a special case of the Alphard iteration construct; it behaves as you would expect a while to behave. A more general iteration mechanism, which allows the author of a form to specify how iterations involving objects of that type are carried out, is described in [Shaw76b].

3. (while through then to while)
   Show: $tips(t) = count - 1 + tips(x) + SIGMA_{u \in s} tips(u) \wedge$
   $(s \neq nullseq \vee x \neq nil) \wedge x = nil \supset$
   $tips(t) = count + 1 - 1 + tips(last(s)) + SIGMA_{u \in leader(s)} tips(u)$
   Proof: x=nil means s≠nullseq whence last(s) and leader(s) are defined (i.e. the pre conditions for top and pop are satisfied). x=nil also means tips(x)=1. The conclusion follows by the definition of SIGMA.

4. (while through else to while)
   Show: $tips(t) = count - 1 + tips(x) + SIGMA_{u \in s} tips(u) \wedge$
   $(s \neq nullseq \vee x \neq nil) \wedge x \neq nil \supset$
   $tips(t) = count - 1 + tips(leftson(x)) + SIGMA_{u \in s \sim rightsubn(x)} tips(u)$
   Proof: x≠nil means the pre conditions of both left(x) and right(x) are met. x≠nil also means tips(x) = tips(leftson(x)) + tips(rightson(x)). The conclusion follows by the definition of SIGMA. It remains to show that the pre condition of push is met. To do this it is convenient to add two terms to the while assertion:
   $length(s) + height(x) \leq height(t)$
   $s = \langle s_1, \ldots, s_k \rangle \wedge 1 \leq j \leq k \supset j + height(s_j) \leq height(t)$
   Assuming these two terms are indeed invariants (proof omitted), the pre condition is met because x≠nil means $height(x) \geq 1$, i.e. $length(s) < height(t)$.

QED

Further Parameterization of Forms

The "<>" notation used above is actually much more broadly applicable than might be suggested by the stack example. To see this, and to motivate another related facility, we shall turn away from the form concept for a moment and consider the more traditional functional abstractions provided by subroutines. Suppose that we wished to write a subroutine which tested for the equality of two vectors. Using a pseudo-Alphard notation such a subroutine might appear as:

```
function eqvecs(A,B:vector(integer,1,10)) returns (eq:boolean) =
  begin
  for i from 1 to 10 do
      if A[i] ≠ B[i] then (eq←false; return);
  eq←true;
  end
```

(This example is not "real" Alphard because of the iteration statement; the companion

paper [Shaw76b] defines the Alphard iteration construct and presents this example in its correct form.)

Much as with the stack example, this program is quite unsatisfying. We would at least like to be able to write a function that would cover a broader class of vectors -- say those of arbitrary length. Unless we do this we will be forced to write a different subroutine for each possible vector length.[12] But even if we were to accommodate different lengths, we might still have to write different subroutines for each possible element type. Once again, if we examine the proof of this subroutine we will find that the only dependence on the element type is the existence of a not-equal operation.

The correctness of the implementation of any parameterized abstraction depends on certain properties of the parameters and is completely independent of others. An abstract "eqvecs" subroutine *should* require that: (1) its two parameter vectors be the same length, (2) the elements of both vectors be the same type, and (3) the type of the elements provide a not-equal operation. It *should not* require that: (1) the vectors be of some pre-specified length, (2) the upper and/or lower bounds of these vectors have some pre-specified value, or (3) the elements have any other properties.

The "<>" notation provides a means of specifying the *required* properties of actual parameters. We shall now introduce *questionmark identifiers* to permit the specification of non-requirements. *Defining occurrences* of such identifiers consist of a "?" immediately followed by an identifier, e.g., "?xyz"; they appear in formal parameter lists and are assigned meaning from the corresponding actual parameters. Multiple occurrences of the same ?identifier are required to have the same meaning in the same scope. *Applied occurrences* of these identifiers are uses of the identifiers without question marks. These may appear anywhere in the scope of their definition -- thus, for example, they may be used to declare variables of the same type as an actual parameter[13].

The use of both the "<>" notation and ?identifiers is illustrated by the following pseudo-Alphard coding of the "eqvecs" subroutine. (The syntax of the iteration statement still prevents this from being proper Alphard.)

---

[12] Such a restriction is one of the less pleasing aspects of Pascal [Habermann73, Wirth75].

[13] There are somewhat pathological situations involving recursive procedures in which this scheme will not work; in particular in these cases it is not possible to determine the proper types at compile time. We choose to ignore these pathologies here.

```
function eqvecs(A,B:vector(?t<≠>,?lb,?ub)) returns (eq:boolean) =
  begin
  for i from lb to ub do
      if A[i] ≠ B[i] then (eq←false; return);
  eq←true;
  end
```

Note that in this implementation the symbols lb and ub appear as applied occurrences in the for statement. The intent is that, whatever the lower and upper bounds of the actual-parameter vectors, these values will be used as the initial and final values of the for statement range. Also, note that the form of the formal parameter list ensures that the two actual parameters will have the same element types and bounds.

We shall not prove this implementation of "eqvecs" (the verification of the true Alphard version appears in [Shaw76b]), but the reader should readily be able to visualize such a proof and to see that it has not been affected by the generalizations introduced.

Protection and Access Control

The "<>" notation introduced above is clearly an extension of the familiar notion of type checking in programming languages; in this section we shall try to show its relation to the protection facilities of modern operating systems, especially those using the *capability* based protection model. In the foregoing discussion we stressed the restrictions imposed on actual parameters by the appearance of the "<>" notation in a formal parameter list. We did not discuss either the restrictions it imposes on the body of the subroutine (or form) or the precise nature of what may appear between the angle-brackets. Those issues will be treated here as well.

Note that "x:X<p>" appearing in a formal parameter list is intended to assert that the body depends on property p, and *only* on property p, of the parameter. Now, from our earlier discussion we know that the only visible properties of an abstraction are those specified in its specifications part. Thus we require that the name "p" be one of the names defined in the specifications part of the form X. Furthermore, since the abstraction being defined claims to depend *only* on the property p, we shall restrict the body of the abstraction to use only this property. That is, all qualifications of x other than "x.p" (or p(x,...)) are illegal. (Note that this is a purely syntactic, compile-time, check. Also note that we must check that any functions called by the body of the abstraction, where x is a parameter to that function, must also require no more than "p" access to it.)

In the terminology of operating systems the specifications part of a form defines a

set of *accesses* to *objects* of the type defined by the form. The "<>" notation defines both the access rights *required* of the actual parameter and *allowed* to the body. Once the actual parameter has been bound to the formal at execution time the formal becomes the name of a *capability* [Fabry74, Graham72, Jones73, Jones74, Lampson71] for the actual. At compile time the formal parameter specification may be viewed as a *template* [Wulf74] for legal actuals.

The analogy with the capability-based model of protection is not yet complete. In an operating system it is generally possible to *restrict* access rights; the "<>" notation permits us to do this at formal/actual parameter binding, but may also be useful in other contexts. For verification purposes, for example, it may be convenient to know that in some block no side-effect producing operations are applied to a specific variable.

A full treatment of a mechanism which provides this type of protection may be found in [Jones76]. For our present purposes we shall simply note that the "<>" notation is permitted in several additional contexts, two of which are discussed below, and in these contexts imply only a rights restriction (not also a requirement as in formal parameter specifications). These contexts are declarations and actual parameters. Consider the declaration:

```
local i:integer<+,-,=,←>;
```

This declaration defines a variable of type integer to which only the operations "+","-", "=", and "←" may be applied. Any other operations defined by the integer form will be illegal -- specifically such things as "*", "/", and relational tests. Such a declaration might be used for a variable which is intended only for use as a counter, for example.

By attaching a rights restriction to the actual parameter of a subroutine invocation the user may ensure that only certain operations are applied by the subroutine. Thus, in the program:

```
begin
local i:integer;
. . .
f(i<+,-,*>);
. . .
end;
```

the main program has all access rights to the variable "i", but restricts the operations that may be performed by "f" to those listed. This is, perhaps, a somewhat strained example since the more common case will be to restrict side-effect producing operations; hopefully, however, it illustrates the point. Once again let us emphasize that this is a purely static, compile-time check. At compile time, the rights *permitted* by the actual parameter are compared to those *required* by the formal; if the former are not a superset of the latter a compile-time error message is generated. There is no run-time overhead.

Now let's turn to the question of what may be written between the angle brackets, especially in the context of a formal parameter specification. To this point we have simply written the *name* of a property, which is generally a function name. This is sufficient in the cases where the type of the formal is specified, but not when the type is characterized by a identifier. Consider an example which involves less suggestive names than those used previously:

```
function f(a:?T<h>)= . . .;
```

The intent is, as before, that the function "f" depend only on the fact that the actual parameter be of a type which provides an "h" operation, *not* its name. But suppose that the type of the actual parameter does provide an operation named "h", but it has nothing to do with the operation which the writer of "f" had in mind. In fact, the writer of "f", or alternatively the correctness of "f", depends on some input-output relation of the "h" operation. Thus, we permit properties appearing in the angle brackets to be described in exactly the same manner as properties appearing in the specifications part of a form definition. For example,

function f(a:?T<h(T,integer) returns (b:boolean) pre β_1 post β_2>)= . . .;

When such specifications appear the problem of validating the legality of an actual parameter is more complex than previously. We must not only establish that the form defining the type of the actual parameter provides a property named "h", but also that: (1) its parameters and result are of the appropriate type and (2) that the precondition required in the specification of that property is implied by β_1 and that the postcondition of that property is sufficient to imply β_2. We do not foresee this proof as part of the compilation process, but rather as another prcof required in the verification of the program.

Another Example: Queues

As a further illustration of both the Alphard language and the verification technique, we now present another example. The example is a finite capacity fifo queue; in all respects but one it is similar to the stack presented earlier. The important difference is that the representation of a given queue configuration is not unique; that is, there may be several concrete representations for the same abstract object. We present one program and its verification with little comment; we then present another implementation of the same specifications.

The specifications describe the behavior of queues in terms of sequences. Queues

are implemented using a vector to record the entries and integers to indicate the front, back, and current length. The enqueue operator, "enq", extends the queue toward higher-indexed vector elements, wrapping around to the zeroth element when the indices are exhausted. The dequeue operation, "deq", returns and removes elements in the order in which they were inserted. The function "size" returns the current queue size.

```
form fifo(T:form<←>, n:integer)=
  beginform

  specifications
    requires n>0;
    let fifo = < . . . xi . . . > where xi is T;
    invariant 0≤length(fifo)≤n;
    initially fifo=nullseq;
    function
      enq(q:fifo, x:T)  pre 0≤length(q)<n  post q=q'~x
      deq(q:fifo) returns x:T
            pre  0<length(q)≤n  post x=first(q') ∧ q=trailer(q');
      size(q:fifo) returns x:integer post x=length(q')

  representation
    unique v:vector(T,0,n-1), f,b,num:integer init (f←num←0; b←n-1);
    rep(v,f,b,num) = if num=0 then <> else
          if f≤b then seq(v,f,b) else seq(v,f,n-1)~seq(v,0,b);
    invariant 0≤num≤n ∧ 0≤f≤n-1 ∧ 0≤b≤n-1 ∧
          (num=0 ∧ n=(b+n-f)mod n + 1 ∨ num>0 ∧ num=(b+n-f)mod n + 1);
    states
      mt when num=0,
      normal when 0<num<n,
      full when num=n,
      err otherwise;

  implementation
    body enq out (q.b=(q.b'+1)mod n ∧ q.v=α(q.v',q.b,x) ∧ q.num=q.num'+1) =
      mt,normal:: (q.b←(q.b+1)mod n; q.v[q.b]←x; q.num←q.num+1);
      otherwise:: FAIL;

    body deq out (q.f=(q.f'+1)mod n ∧ x=q.v'[q.f'] ∧ q.num=q.num'-1) =
      normal,full:: (x←q.v[q.f]; q.f←(q.f+1)mod n; q.num←q.num-1);
      otherwise:: FAIL;
```

```
body size out (x=q.num') =
    mt,normal,full:: x←q.num;
    err:: FAIL;

endform
```

To save space and reduce clutter, the proof omits from I_c the two terms $0 \le f \le n-1$ and $0 \le b \le n-1$. That they are part of I_c follows because of the mod operations in the bodies of enq and deq and because of init. The requires clause $n>0$ guarantees disjoint states and also makes the "mod n" operation well-defined. As in the istack proof, verifications involving FAIL are omitted.

For the form

1. Representation validity
   Show: $0 \le num \le n \wedge (num=0 \wedge n=(b+n-f) \bmod n + 1 \vee num>0 \wedge num=(b+n-f) \bmod n + 1) \supset 0 \le length(rep(x)) \le n$
   Proof: $0 \le num \le n$, so the conclusion follows by showing length(rep(x)) = num. First, $num=0 \supset length(\langle\rangle)=0=num$. Second, $f \le b \wedge num>0 \supset length(seq(v,f,b)) = b-f+1 = (b+n-f) \bmod n + 1 = num$. Third, $f>b \wedge num>0 \supset length(seq(v,f,n-1) \sim seq(v,0,b)) = (n-f)+(b+1) = (b+n-f) \bmod n + 1 = num$.

2. Initialization
   Show: $n>0$ { f←num←0; b←n-1 } $rep(v,0,n-1,0)=nullseq \wedge I_c$
   Proof: $rep(v,0,n-1,0) = \langle\rangle$, i.e., nullseq. For I_c note that the first term of the or holds for both n=1 and n>1

As convenient notation below, let z = (q.v, q.f, q.b, q.num). Furthermore, steps 4b in the proof are simplified if we rewrite the rep function. Define seqm(v, f, b, n) to be the sequence

$$\langle v_f, v_{(f+1) \bmod n}, v_{(f+2) \bmod n}, \cdots, v_b \rangle$$

i.e., the indices are computed mod n (the "m" in seqm suggests "mod"). Then rep(v,f,b,num) = if num=0 then <> else seqm(v,f,b,n). To see that this is the same as the original rep function, first note that $0 \le f \le n-1$ and $0 \le b \le n-1$. If num=0 it is clear. If $f \le b$ then $(f+i) \bmod n = f+i$ for $1 \le i \le b-f$ so seq(v,f,b) = seqm(v,f,b,n). If $f>b$ let $j=n-f$. Then

$$seq(v,f,n-1) \sim seq(v,0,b) = seqm(v,f,n-1,n) \sim seqm(v,(f+j) \bmod n,b,n) = seqm(v,f,b,n)$$

For the function enq

3. Concrete operation

Show: $0 \le q.num < n \wedge I_c$ { q.b←(q.b+1)mod n; q.v[q.b]←x; q.num←q.num+1 } $\beta_{out} \wedge I_c$

Proof: β_{out} is clear. $0 \le q.num < n \supset 0 \le q.num+1 \le n$. The last term of the or becomes q.num+1 = ((q.b+1)mod n + n-q.f)mod n + 1. If n=1 then q.num=0 and it holds. If n>1 then ((q.b+1)mod n + n-q.f)mod n + 1 = ((q.b+n-q.f)mod n + 1)mod n + 1. If q.num>0 this is q.num mod n + 1 = q.num+1. If q.num=0 this is n mod n + 1 = 1 = q.num+1.

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge 0 \le length(rep(z')) < n \wedge \beta_{out}(z) \supset q = q' \sim x$

Proof: q = rep(z) = seqm(q.v,q.f,(q.b'+1)mod n,n) = seqm(q.v,q.f,q.b',n)~q.v[q.b] = seqm(q.v',q.f',q.b',n)~x = q'~x

For the function deq

3. Concrete operation

Show: $0 < q.num \le n \wedge I_c$ { x←q.v[q.f]; q.f←(q.f+1)mod n; q.num←q.num-1 } $\beta_{out} \wedge I_c$

Proof: β_{out} is clear. $0 < q.num \le n \supset 0 \le q.num-1 \le n$. The rest of I_c follows similarly to enq.3.

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge 0 < length(rep(z')) \le n \wedge \beta_{out}(z) \supset x = first(q') \wedge q = trailer(q')$

Proof: x = q.v'[q.f'] = first(q'). First(q') is defined since length(rep(z'))>0. q = rep(z) = seqm(q.v',(q.f'+1)mod n,q.b',n) = trailer(q')

For the function size

3. Concrete operation

Show: $0 \le q.num \le n \wedge I_c$ { x←q.num } x=q.num' $\wedge I_c$

Proof: clear

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge x = q.num' \supset x = length(q')$

Proof: As in step 1, $I_c \supset length(rep(z)) = num$. Hence x = q.num' = length(rep(z')) = length(q')

QED

Another way to implement a queue is to use a vector(T,0,n) rather than a vector(T,0,n-1). The integer to indicate current length can be eliminated because now front and back are sufficient. The specifications part is unchanged; the representation and implementation parts do change in various ways. Accordingly, the proof of the form will change in each of the four steps. The modified proof steps are similar to the previous ones, perhaps even easier because I_c is much simpler. The previous proofs provide useful guidance, at least to a human. What does not change, of course, is a proof that uses the fifo form because the specifications are identical. The modified form and its proof are given next. Here z= (q.v, q.f, q.b).

```
form fifo(T:form<←>, n:integer)=
  beginform

  specifications (identical to the original fifo form)
     requires n>0;
     let fifo = < . . . xi . . . > where xi is T;
     invariant 0≤length(fifo)≤n;
     initially fifo=nullseq;
     function
        enq(q:fifo, x:T)  pre 0≤length(q)<n  post q=q'~x
        deq(q:fifo) returns x:T
                pre  0<length(q)≤n  post x=first(q') ∧ q=trailer(q');
        size(q:fifo) returns x:integer post x=length(q')

  representation
     unique v:vector(T,0,n), f,b:integer init (f←0; b←0);
     rep(v,f,b) = if f=(b+1)mod(n+1) then <> else seqm(v,f,b,n+1)
     invariant 0≤f≤n ∧ 0≤b≤n
     states
        mt when f=(b+1)mod(n+1),
        full when f=(b+2)mod(n+1),
        normal otherwise;

  implementation
     body enq out (q.b=(q.b'+1)mod(n+1) ∧ q.v=∝(q.v',q.b,x)) =
        mt,normal:: (q.b←(q.b+1)mod(n+1); q.v[q.b]←x);
        otherwise:: FAIL;
```

```
body deq out (q.f=(q.f'+1)mod(n+1) ∧ x=q.v'[q.f']) =
    normal,full:: (x←q.v[q.f]; q.f←(q.f+1)mod(n+1));
    otherwise:: FAIL;

body size out (x=(q.b'-q.f'+n+2)mod(n+1)) =
    x←(q.b-q.f+n+2)mod(n+1);

endform
```

For the form

1. Representation validity
   Show: $0 \le f \le n \wedge 0 \le b \le n \supset 0 \le length(rep(x)) \le n$
   Proof: Length(<>) = 0. Length(seqm(v,f,b,n+1)) = (b-f+1+n+1)mod(n+1) so $0 \le length(rep(x)) \le n$. (If $f \ne (b+1) \bmod (n+1)$ then $0 \ne length$)

2. Initialization
   Show: $n>0$ { f←0; b←n } $rep(v,0,n)=nullseq \wedge I_c$
   Proof: Since $0=(n+1) \bmod (n+1)$, $rep(v,0,n)=<>$. I_c is clear.

The three concrete operations (steps 3) are clear. The three steps 4a follow since each β_{in} is true.

For the function enq

4b. β_{post} holds
   Show: $I_c \wedge 0 \le length(rep(z')) < n \wedge \beta_{out}(z) \supset q = q' \sim x$
   Proof: q = rep(z) =seqm(q.v,q.f,(q.b'+1)mod(n+1),n+1) =
   seqm(q.v,q.f,q.b',n+1)~q.v[q.b] = seqm(q.v',q.f',q.b',n+1)~x = q'~x

For the function deq

4b. β_{post} holds
   Show: $I_c \wedge 0 < length(rep(z')) \le n \wedge \beta_{out}(z) \supset x = first(q') \wedge$
   q = trailer(q')
   Proof: x = q.v'[q.f'] = first(q'). First(q') is defined since
   length(rep(z'))>0. q = rep(z) = seqm(q.v',(q.f'+1)mod(n+1),q.b',n+1) =
   trailer(q')

For the function size

4b. β_{post} holds
   Show: $I_c \wedge x=(q.b'-q.f'+n+2) \bmod (n+1) \supset x=length(q')$
   Proof: x = (q.b'-q.f'+n+2)mod(n+1) = length(rep(z')) = length(q')

QED

For verifications involving FAIL, it is convenient to use the facts

length(rep(v,f,b))=0 iff f=(b+1)mod(n+1)
length(rep(v,f,b))=n iff f=(b+2)mod(n+1)

Conclusion

We have described the data abst action facilities of the Alphard language and the associated verification methodology. In this conclusion we shall attempt to allay some fears which our programming colleagues may have after reading this paper, and then we shall return to the issue raised in the introduction: the symbiotic effect of methodological and verification concerns in the design process.

Much of the effort expended in the design of programming languages over the past fifteen years has been aimed at improving the convenience with which a programmer may express his algorithms. Alphard in some ways represents the antithesis of this trend. In general, for example, Alphard programs are somewhat longer than similar Fortran or Algol programs. (The size increase seems to result mainly from the requirement that specification and verification information be supplied. Several of our examples, such as the "simpleset" form developed in [Shaw76b] and the tree manipulation program in [Shaw76a], suggest further that the growth may be illusory.) We believe this expansion to be completely acceptable for several reasons:

1. It is not clear that the concern for convenience has in fact saved programmers much work. Although isolated examples of the utility of elaborate features, e.g., array manipulation in PL/I, may be found, the data on actual language usage [Alexander72, Knuth71, Wichmann70,73] suggest that these features are so rarely used that the labor saved is vanishingly small.

2. Actual coding generally represents only a small fraction of the total effort expended on a project (e.g., 15-25% or less), whereas debugging, system integration, and testing represent a large fraction (e.g., 30-50% or more)[Goldberg73]. Thus, even if we were to double coding time (which we do *not* believe will happen) but in the process could halve the other times, total project time could be reduced. Alphard addresses primarily the latter costs. Suppose we were to change the representation and implementation, but not the specifications, of the stack form. The form itself would have to be reverified, but the programs using it (e.g., tipcount) and the verifications of those programs would remain unchanged.

3. We hope that with a language such as Alphard, the promise of extensible languages will be realized -- that a library of useful abstractions will develop, and that programmers will thus simply not have to program as much to get a new system. Although the notion of program libraries is an old one, it seems (to us) to have had less impact than the notion warrants. Our hypothesis is that the availability of verified abstractions in the library will change this, but that hypothesis cannot be tested yet.

We appreciate that there is considerable scepticism in the programming community concerning the practical applicability of verification techniques. This scepticism extends to both automated verification aids (e.g., theorem provers) and the ability of "typical" programmers to write the requisite formal specifications. To the first concern we cite the accomplishments of existing verification systems [Good75, vonHenke75, Suzuki75]. All the examples in this paper and in [Shaw76a, 76b] appear generally within the capabilities of these systems. As for the second issue, two of the authors (Wulf and Shaw) are primarily programmers, not verifiers; on the basis of our experience thus far we all believe that the formulation of the specifications is a learnable formalization of what systems analysts do anyway. We believe the potential gains more than justify the training required.

The practical programmer may also question the potential (in)efficiency of Alphard programs; the pragmatic programmer who has experimented with some of the newer "high level" languages has ample cause to ask such a question. The intended application area for Alphard includes large systems programs where efficiency is often essential. We believe *very* efficient code can be compiled for Alphard programs, although the nature of this paper and the material presented do not tend to support this position. We can at present defend our belief with only one observation on the present discussion. In typical high level languages the compiler-writer makes certain implementation decisions (for example, how arrays will be represented); since these decisions are irrevocable, the programmer cannot choose representational optimizations which will make a particular program more time or space efficient. The usual argument is that these decisions must be made by the compiler-writer to prevent the programmer from making a mistake and hurting himself. Alphard takes a totally different position: all such decisions may be made by the programmer (if he chooses), but we do demand that he verify that they are correct. (That's why names such as "integer" and "vector" are considered as simply ordinary form names which are provided by a standard prelude.) In effect, we have no objection to *dirty* coding tricks so long as they are correct and can be verified.

Now let us return to the interaction of verification and methodology in our design. It is perhaps simplistic to observe that the things which are easily understood (that is, the things which we can informally convince ourselves are true) are usually easy to prove formally. Conversely, the things which are familiar or admit of a simple formal description tend to be easy to understand. This observation is the basis of our remarks.

During the design of Alphard, we repeatedly proposed "features" which either were difficult to formulate proof rules for or which looked suspect on methodological grounds. We usually found that such a problem signalled an unforeseen problem in the other domain. For example, our original for statement was much more elaborate than the one described in [Shaw76b], but seemed plausible on methodological grounds. Its verification, however, was a horror to behold. Subsequently we have become convinced that the complexity of its verification was symptomatic of a difficulty which any reader would have in attempting to understand the statement or its use.

Conversely, good ideas in one domain generally proved to be good in the other as well. The whole form concept, for example, was introduced for methodological reasons. It is this factorization and isolation, however, which appears to make either hand or mechanical verification feasible. Similarly, the notion of generators as described in [Shaw76b] was introduced on methodological grounds, but is simplifying the verification of many loops. Since loop control is implicit rather than explicit, one verification of that loop control suffices. Various predicates were introduced because they were needed for verification, but their presence seems to direct our thinking toward things which, on methodological grounds, we ought to worry about. Finally, the explication of the verification technique exposed the need for certain features, e.g., the init clause in the representation part, which at best were thought of as conveniences and at worst would have been missed completely on the basis of methodological and/or language considerations alone. Dijkstra [Dijkstra75] describes in general terms related experiences.

One closing point: Alphard has not yet been implemented. Although an implementation is now underway, the authors and their colleagues made an early and conscious decision not to implement too early, thereby avoiding premature commitment to design decisions. Though we may have frustrated some of our colleagues at other research institutions by changing the language almost daily, we believe this has been the right approach. We hope, but will not promise, that the publication of this document and of [London76, Shaw76a, 76b] represents a stable point in those features of the language which have been discussed.

Acknowledgements

We owe a great deal to our colleagues at CMU and ISI, especially: Mario Barbacci, Donald Good, John Guttag, Paul Hilfinger, David Jefferson, Anita Jones, David Lamb, David Musser, Karla Perdue, Kamesh Ramakrishna, and David Wile. We would also like to thank James Horning and Barbara Liskov and their groups at the University of Toronto and Massachusetts Institute of Technology, respectively, for their critical reviews of Alphard. Finally, we very much appreciate the perceptive responses that a number of our colleagues have made on an earlier draft of this paper.

References

[Alexander72] William Gregg Alexander, "How A Programming Language is Used", *CSRG Technical Report 10*, University of Toronto, February 1972.

[Baker72] F. T. Baker, "Chief Programmer Team Management of Programming", *IBM Systems Journal*, 11, 1, 1972 (pp. 56-73).

[Brooks75] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975.

[Burstall74] R. M. Burstall, "Program Proving as Hand Simulation with a Little Induction", *Proc. of IFIP Congress 74*, 1974 (pp. 308-312).

[Buxton70] J. N. Buxton and B. Randell (eds.), *Software Engineering Techniques, Report on A Conference Sponsored by the NATO Science Committee, Rome, Italy, October 27-31, 1969*, NATO, April 1970.

[Dahl72] Ole-Johan Dahl and C. A. R. Hoare, "Hierarchical Program Structures", in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare), Academic Press, 1972 (pp. 175-220).

[DataConference76] *Proc. of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure* and *Supplement to the Proc.*, March 1976.

[Dijkstra68a] E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness", *BIT*, 8, July 1968 (pp. 174-186).

[Dijkstra68b] Edsger W. Dijkstra, "Go To Statement Considered Harmful", *Communications of the ACM*, 11, 3, March 1968 (pp. 147-148).

[Dijkstra72] Edsger W. Dijkstra, "Notes on Structured Programming", in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare), Academic Press, 1972 (pp. 1-82).

[Dijkstra75] Edsger W. Dijkstra, "Correctness Concerns and, Among Other Things, Why They Are Resented", *Proc. International Conference on Reliable Software*, April 1975 (pp.546-550).

[Fabry74] R. S. Fabry, "Capability-Based Addressing", *Communications of the ACM*, 17, 7, July 1974 (pp. 403-412).

[Floyd67] Robert W. Floyd, "Assigning Meanings to Programs", *Proc. Symp. in Applied Mathematics, Vol 19* (J. T. Schwartz, ed.), American Mathematical Society, 1967 (pp. 19-32).

[Geschke75] C. M. Geschke and J. G. Mitchell, "On the Problem of Uniform References to Data Structures", *IEEE Transactions on Software Engineering*, SE-1, 2, June 1976 (pp. 207-219).

[Goldberg73] Jack Goldberg (ed.), *Proc. of a Symposium on the High Cost of Software*, SRI, September 1973.

[Good75] Donald I. Good, Ralph L. London, and W. W. Bledsoe, "An Interactive Program Verification System", *IEEE Transactions on Software Engineering*, SE-1, 1, March 1975 (pp. 59-67).

[Graham72] G. Scott Graham and Peter J. Denning, "Protection -- Principles and Practice", *Proc. Spring Joint Computer Conference*, 1972 (pp. 417-429).

[Gries74] David Gries, "On Structured Programming - A Reply to Smoliar", *ACM Forum, Communications of the ACM*, 17, 11, November 1974 (pp. 655-657).

[Guttag75] John V. Guttag, "The Specification and Application to Programming of Abstract Data Types", (Ph. D. Thesis), *CSRG Technical Report 59*, University of Toronto, September 1975.

[Guttag76a] John Guttag, "Abstract Data Types and the Development of Data Structures", *Supplement to the Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure*, March 1976 (pp. 37-46).

[Guttag76b] John V. Guttag, Ellis Horowitz, and David R. Musser, "Abstract Data Types and Software Validation", *USC Information Sciences Institute Technical Report*, 1976.

[Habermann73] A. N. Habermann, "Critical Comments on the Programming Language Pascal", *Acta Informatica*, 3, 1, 1973 (pp. 47-57).

[vonHenke75] F. W. von Henke and David C. Luckham, "A Methodology for Verifying Programs", *Proc. International Conference on Reliable Software*, April 1975 (pp. 156-164).

[Hoare69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, 12, 10, October 1969 (pp. 576-580, 583).

[Hoare72a] C. A. R. Hoare, "Notes on Data Structuring", in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare), Academic Press, 1972 (pp. 83-174).

[Hoare72b] C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica*, 1,4, 1972 (pp. 271-281).

[Jones73] Anita Katherine Jones, "Protection in Programmed Systems", (Ph.D. Thesis), *Carnegie-Mellon University Technical Report*, 1973.

[Jones74] Anita K. Jones and William A. Wulf, "Towards the Design of Secure Systems", *Software-Practice and Experience*, 5, 4, 1975 (pp. 321-336).

[Jones76] Anita K. Jones and Barbara H. Liskov, "An Access Control Facility for Programming Languages", *Computation Structures Group Memo 137*, Massachusetts Institute of Technology and *Carnegie-Mellon University Technical Report*, 1976.

[Knuth71] Donald E. Knuth, "An Empirical Study of Fortran Programs", *Software-Practice and Experience*, 1, 1971 (pp. 105-133).

[Lampson71] Butler Lampson, "Protection", *Fifth Princeton Conference on Information Sciences and Systems*, 1971 (pp. 437-443).

[Liskov74] Barbara Liskov and Stephen Zilles, "Programming with Abstract Data Types", *SIGPLAN Notices*, 9, 4, April 1974 (pp. 50-59).

[Liskov75a] Barbara Liskov, "A Note on CLU", *MAC-TR*, Massachusetts Institute of Technology, June 1975.

[Liskov75b] B. H. Liskov and S. N. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering*, SE-1, 1, March 1975 (pp. 7-19).

[London75] Ralph L. London, "A View of Program Verification", *Proc. International Conference on Reliable Software*, April 1975 (pp. 534-545).

[London76] Ralph L. London, Mary Shaw, and Wm. A. Wulf, "Abstraction and Verification in Alphard: A Symbol Table Example", *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Manna74] Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[Naur66] Peter Naur, "Proof of Algorithms by General Snapshots", *BIT*, 6,4, 1966 (pp. 310-316).

[Naur69] Peter Naur and Brian Randell (eds.), *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, October 7-11, 1968*, NATO, January 1969.

[Parnas71] D. L. Parnas, "Information Distribution Aspects of Design Methodology", *IFIP Congress 1971*, Booklet TA-3 (pp. 26-30).

[Parnas72a] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15, 12, December 1972 (pp. 1053-1058).

[Parnas72b] D. L. Parnas, "A Technique for Software Module Specification with Examples", *Communications of the ACM*, 15, 5, May 1972 (pp. 330-336).

[Ross70] Douglas Ross, "Uniform Referents: An Essential Property for a Software Engineering Language", in *Software Engineering, Volume 1* (J. J. Tou, ed.), Academic Press 1970 (pp. 91-101).

[Schuman71] Stephan A. Schuman (ed.), "Proceedings of the International Symposium on Extensible Languages", *SIGPLAN Notices*, 6, 12, December 1971.

[Shaw76a] Mary Shaw, "Abstraction and Verification in Alphard: Design and Verification of a Tree Handler", *Carnegie-Mellon University Technical Report*, 1976.

[Shaw76b] Mary Shaw, Wm. A. Wulf, and Ralph L. London, "Abstraction and Verification in Alphard: Iteration and Generators", *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Spitzen75] Jay Spitzen and Ben Wegbreit, "The Verification and Synthesis of Data Structures", *Acta Informatica*,4,2, 1975, (pp. 127-144).

[Suzuki75] Norihisa Suzuki, "Verifying Programs by Algebraic and Logical Reduction", *Proc. International Conference on Reliable Software*, April 1975 (pp. 473-481).

[Wegbreit76] Ben Wegbreit and Jay M. Spitzen, "Proving Properties of Complex Data Structures", *Journal of the ACM*, 23, 2, April 1976 (pp. 389-396).

[Weinberg71] Gerald M. Weinberg, *The Psychology of Computer Programming*, van Nostrand Reinhold, 1971.

[Whorf56] Benjamin Lee Whorf, "A Linguistic Consideration of Thinking in Primitive Communities", in *Language, Thought, and Reality* (John B. Carroll, ed.), MIT Press, 1956.

[Wichmann70] B. A. Wichmann, "Some Statistics from Algol Programs", *Central Computer Unit Report 11*, National Physical Laboratory, August 1970.

[Wichmann73] B. A. Wichmann, "Basic Statement Times for Algol 60", *NPL Report NAC 42*, National Physical Laboratory, November 1973.

[vanWijngaarden69] A. van Wijngaarden (ed.), B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, "Report on the Algorithmic Language Algol 68", *Numerische Mathematik*, 14, 1969 (pp. 79-218).

[Wirth71] Niklaus Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, 14, 4, April 1971 (pp. 221-227).

[Wirth75] Niklaus Wirth, "An Assessment of the Programming Language Pascal", *IEEE Transactions on Software Engineering* SE-1, 2, June 1975 (pp.192-198).

[Wulf72] W. Wulf and Mary Shaw, "Global Variables Considered Harmful", *SIGPLAN Notices*, 8, 2, February 1973 (pp. 28-34).

[Wulf74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The Kernel of a Multip ocessor Operating System", *Communications of the ACM*, 17, 6, June 1974 (pp. 337-345).

[Zilles75] S. N. Zilles, "Abstract Specifications for Data Types", IBM Research Laboratory, San Jose, January 1975.

Appendix A
Formal Definition of a Sequence

In the examples presented in the body of the paper the notion of a (mathematical) sequence was used several times. An axiomatic definition of a sequence may be found in [Hoare 72a]; a version adapted to the needs of our examples is included below for completeness.

1. Let D be a set called the *domain* of the elements of a sequence; then
   (a) <> is a sequence, the *null sequence*, sometimes denoted "nullseq".
   (b) if x is a sequence and d∈D, then x~<d> is a sequence
   (c) the only sequences are those specified by (a) & (b)

2. The following functions and relations are defined:
   (a) $last(x\sim\langle d\rangle) =_{df} d$
   (b) $leader(x\sim\langle d\rangle) =_{df} x$
   (c) $x\sim(y\sim z) =_{df} (x\sim y)\sim z$
   (d) $first(\langle d\rangle) =_{df} d$
   $x\neq\langle\rangle \supset first(x\sim\langle d\rangle) =_{df} first(x)$
   (e) $trailer(\langle d\rangle) =_{df} \langle\rangle$
   $x\neq\langle\rangle \supset trailer(x\sim\langle d\rangle) =_{df} trailer(x)\sim\langle d\rangle$
   Note: "first", "last", leader, and "trailer" are not defined on the null sequence, <>.
   (f) $length(\langle\rangle) =_{df} 0$
   $length(x\sim\langle d\rangle) =_{df} 1+length(x)$

3. The notation $\langle d_1,d_2,\ldots,d_n\rangle$ is an abbreviation for $\langle\rangle\sim\langle d_1\rangle\sim\langle d_2\rangle\sim\ldots\sim\langle d_n\rangle$.

4. If V is a vector whose elements are in D and n and m are integers, then "seq(V,n,m)" is an abbreviation defined by:
   $n>m \supset seq(V,n,m) =_{df} \langle\rangle$
   $n\leq m \supset seq(V,n,m) =_{df} \langle V_n,V_{n+1},\ldots,V_m\rangle$

5. The definition of equality of sequences is included in 1 and 2 as the two theorems:
   $x=y$ iff $(x=y=\langle\rangle \vee first(x)=first(y)\wedge trailer(x)=trailer(y))$
   $x=y$ iff $(x=y=\langle\rangle \vee last(x)=last(y)\wedge leader(x)=leader(y))$