ABSTRACTION and VERIFICATION in ALPHARD: Iteration and Generators

Mary Shaw, Carnegie-Mellon University

Wm. A. Wulf, Carnegie-Mellon University

Ralph L. London, USC Information Sciences Institute

August 20, 1976

Abstract: The Alphard <u>form</u> provides the programmer with a great deal of control over the implementation of abstract data types. In this report we extend the abstraction techniques from simple data representation and function definition to the *iteration statement*, the most important point of interaction between data and the control structure of the language itself. We introduce a means of specializing Alphard's loops to operate on abstract entities without explicit dependence on the representation of those entities. We develop specification and verification techniques that allow the properties of such iterations to be expressed in the form of proof rules. We also provide a means of showing that a generator will terminate and obtain results for common special cases that are essentially identical to the corresponding constructs in other languages.

Keywords and Phrases: abstraction and representation, abstract data types, assertions, control specialization, correctness, generators, invariants, iteration statements, modular decomposition, program specifications, programming languages, programming methodology, proofs of correctness, 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. This report is one in a series being printed jointly by CMU and ISI.

Contents

Introduction

This paper is one in a series describing the Alphard programming system and its associated verification methods. It presumes that the reader is familiar with the material in [Wulf76a,b], particularly the use of forms for abstraction and the verification methodology for forms.

The primary goal of the form mechanism is to permit and encourage the localization of information about a user-defined *abstraction*. Specifically, the mechanism is designed to localize both verification and modification. Other reports on Alphard have discussed ways to isolate specific information about representation and implementation; in this paper we deal with localizing another kind of information.

Suppose that S is a "set-of-integers" and that we wish to compute the sum of the integers in this set. In most contemporary programming languages we would have to write a statement such as

sum ← 0; for i ← 1 step 1 until S.size do sum ← sum + S[i]

or possibly

p ← S; sum ← 0; while p ≠ nil do (sum ← sum + p.value; p ← p.next)

or, if we know that the set elements all lie in the range [lb..ub], then we might write

sum←0; for i←lb to ub do if i ∈ S then sum←sum+i

None of these statements is really satisfactory. First, they all seem to imply an order to the summation, whereas the abstract computation does not. Next, the first statement strongly suggests a vector implementation of the set and the second a list implementation. (Although other implementations are not excluded, the resulting loops will probably be unacceptably inefficient.) The third statement does not suggest an implementation of the set, but may be too inefficient if the cardinality of the set is much smaller than ub-lb+1.

It would be much better if we could write something like

sum←0; for x∈S do sum←sum+x

which implies nothing about either the order of processing or the representation of sets. Except for notational differences, this latter example illustrates our goal. We want to encourage suppression of the details of how iteration over that abstract data structure is actually implemented. The difficulty in doing this is that the abstract objects are not predefined in Alphard. Hence it is the author of the abstraction who must specify the implementation of (the analog of) "x∈S".

We resolve the problem by separating the responsibility for defining the meaning of a loop into three parts. (1) Alphard defines the (fixed) syntax and the broad outline of the semantics. (2) The definition of the abstraction that is controlling the iteration fills in the details of the loop control (in particular, the algorithms for selecting the next element and terminating the loop). (3) The user supplies the loop body. Conventional languages provide only a small, fixed number of alternatives (usually one) for the second part of this information. In Alphard, it is supplied by the form that defines the abstraction; we say this part of the definition *specializes* the iteration statement to that abstraction. Related constructs appear in IPL-V as generators [Newell64] and in Lisp as the mapping functions [McCarthy62, Weissman67].

One of the major goals of Alphard is to provide mechanisms to support the use of good programming methodology. The rationale for generators given above is based on methodological considerations; that is, it is generally *good* to abstract from the implementation and hide its details. Generators permit us to do this for control constructs much as the functions in a form permit abstraction of operations (see [Wulf76a,b]).

A second major goal is to provide the ability to specify precisely the effect of a program and then prove the program implements that specification. To meet this goal, we must provide more than just the language mechanism for generators: we must also provide both a way to specify their effects and a corresponding proof methodology. A natural means of doing this for generators is somewhat different from one for functions. Functions are naturally characterized by predicates which relate the state of the computation before their invocation to its state afterward. Generators, however, are not *invoked* in the usual sense; rather they are used to control the repeated execution of an arbitrary "body" of an iteration statement. Thus, a natural specification of a generator is in terms of a "proof rule" which permits the effect of the entire iteration statement to be expressed.

This report contains two strongly related components: first we introduce the language mechanism for generators, then we turn to the specification and verification of generators and of the iteration statements which use them. We begin with a digression on a language feature which is not discussed elsewhere, but is needed for the definition of generators. We then introduce the two Alphard iteration statements and show how they can be specialized by the user. One of these is an iteration construct designed for searching a series of values for an element with a desired property. It should replace most of the loop-exit gotos used in current languages. (Interlisp [Teitelman75] contains a wide variety of iteration statements, one of which specializes to this construct.)

We obtain general proof rules for the two loop constructs, then state a series of simplifying assumptions that certain generators may satisfy. We obtain a corresponding series of proof rules whose simplicity increases with the restrictiveness of the assumptions we make about the generators. These assumptions lead both to rules that correspond directly to familiar rules for iteration (e.g., those of Pascal [Hoare73, Jensen74]) and to simple rules for a substantial number of interesting abstract structures (e.g., those given by Hoare [Hoare72a]).

We then show how to use proof rules instead of functional descriptions to specify many of the forms which define generators. We also give a technique for showing that loops using a generator will halt (assuming the loop body terminates). We prove, with one application of this technique, that many common generators have this property.

Finally, we develop an extended example in which a programmer-defined abstraction is treated as primitive in the implementation of another abstraction. A generator defined in the former is used in the implementation and verification of the latter.

Form Extensions

In this section we introduce another language facility which makes it more convenient to define certain abstractions and to manage the definitions after they are written. The facility allows a programmer to define one form as an *extension* of another. The new form will have most or all of the properties of the old one, plus some additional ones. (This mechanism is similar to, and derived from, the *class concatenation* mechanism of Simula [Dahl72].) We introduce this mechanism at this point because it is needed for generator definitions, which will be discussed in the next section.

The following skeletal form definition illustrates most of the major attributes of the extension mechanism:

```
form counter extends i:integer=
  beginform
  specifications
     initially counter = 1;
     inherits < =, ≠, <, >, ≤, ≥ >;
     function
        inc(x:counter) . . .,
        dec(x:counter) . . .;
  representation
     init i←1;
  implementation
     body inc = x.i ← x.i+1;
     body dec = x.i ← x.i-1;
  endform
```

The general flavor of the mechanism is that the new abstraction, "counter" in this case, is to be an extension of a previously defined one called its *base type*, here "integer". As such, the new abstraction inherits the indicated properties specified for the base type, and may appear in contexts where the base type was permitted (e.g., as an actual parameter where the formal specifies the base form). Further, the new abstraction has the additional properties specified in the extension form, "inc" and "dec" in this case.

Even though the newly defined form is an extension of another, the body of the new form is not granted access to the representation of the old one; the only access rights granted to the body of the new form are those defined in the specifications of the one being extended. Thus, although the extension may add (and delete, see below) properties of the extended abstraction, it *cannot* affect the correctness of its implementation, and we need not reverify the properties of the original. (Indeed, since these properties are identical we do not demand that they even be specified.)

In this example, and indeed more generally, it is not desirable for *all* of the properties of the old abstraction to be inherited by the new one. The "<>" notation may be used as in [Wulf76a,b] to list the rights that the instantiation of the new abstraction is *allowed* to inherit. Thus the maximum set of rights permitted to the instantiation of a "counter" is the union of the inherited rights (=,≠,<,>,≤,≥) and the newly defined rights (inc and dec). Note in particular that assignment to a counter is *not* one of the inherited rights; thus the only way to achieve a side-effect on a counter is through the operations "inc" and "dec". The *implementation* of the extension form may, of course, use all operations on the base type.

As a practical matter, the instantiation of the base form ("i" from "i:integer" in this example) may be considered a part of the representation part of the extended form. Note, however, that this need not be the entire representation part of the extension; in many cases the extension will involve additional data.

Iteration Constructs in Alphard

Alphard provides two iteration commands: the for statement is used for iteration over a complete data structure, and the first statement is used (primarily) for search loops. As mentioned above, each of these commands may be *specialized* for each use. Specialization information is provided through a standard interface called a *generator*. A generator is itself simply a form, but it must adhere to certain special requirements that make it mesh with the semantics of iteration statements:

(a) It must provide two functions (named &init and &next) with properties described below.

(b) Invocation of these functions in a prescribed order must produce a sequence of values to bind to the loop variable.[1]

(c) It must be an *extension* whose base type is the same as the type of the elements being supplied to the loop body.

Before we discuss generators intended for specific structures, we will illustrate the use of the for and first statements with simple counting loops.

The for Statement

We shall begin with the for statement. The syntax for the statement is[2]

for x: gen(y) while β(x) do ST(x,y,z)

where β(x) is an expression, the statement ST(x,y,z) is the loop body, x is the instantiation of the generator "gen", y is the set of instantiation parameters to the generator, and z is the set of other variables used in the statement. The phrase "x: gen", which is our notational analog of the "x$\in$S" in the introduction, means "bind x to an instantiation of the generator named gen intended specifically to generate the elements specified by y". Then x may appear free in β and ST; like any loop variable, x is rebound for each pass through the loop.

The meaning of the for loop is given by the statement

```
begin local x: gen(y);
   π ← x.&init;
   while π cand β(x) do
        (ST(x,y,z); π ← x.&next)
end
```

Here, *cand* is the "conditional and" operator: "b_1 *cand* b_2" ≡ "if b_1 then b_2 else false". Also, β and ST are taken from the for statement, and x.&init and x.&next are functions supplied by the generator as described below.[3] The compiler-generated variable, π, is not accessible to the

---

[1] Although we call this a "loop variable", it will not normally be possible to alter its value within the loop body.

[2] Either "for x:gen(y)" or "while β(x)" may be omitted yielding the pure while and pure for statements, respectively. If "while β(x)" is omitted, β is assumed to be identically true. If "for x: gen" is omitted, no x is declared or set, β and ST (clearly) cannot depend on x, and &init and &next are assumed to be the constant true. β may depend on y and z in addition to x.

programmer.

One of the generators defined in the standard prelude is

upto(lb,ub: integer) extends k: integer

This generator produces the sequence of values <lb, lb+1, lb+2, . . . , ub-1, ub>, or the empty sequence if lb>ub. This generator, in combination with the for statement, provides the familiar "stepping" loop found in nearly all programming languages; for example, an Alphard loop for summing the integers from 1 to n is

sum ← 0; for j: upto(1,n) do sum ← sum+j

Note that *two* types are involved in this example. We said in earlier contexts that the notation "j: upto(. . .)" means "bind j to an instantiation of upto". This implies that the type of j is "upto". However, notice that j is used in the body of the loop as though it were an integer. This is possible because of the extension mechanism described in the previous section. Although the apparent type of j is upto, form upto extends integers, inheriting all operations except assignment (the definition is given in the next section). As a result, integer operations on j are legal and behave as expected.

The first Statement

One of the common uses of loops is for searching a sequence of values for the first one which passes some test. The use of an ordinary loop construct for this purpose is probably the most common cause of *necessary* gotos in conventional programming languages: once the test has been satisfied, there is no reason to continue executing the loop. Since this case occurs so often, Alphard provides a special syntax for it. We may write[4]

first x:gen(y) suchthat $\beta(x)$ then $S_1(x,y,z)$ else $S_2(y,z)$

where S_1 and S_2 are statements and β is an expression. Again, x is an instantiation of generator gen and may appear free in β and S_1 (but *not* in S_2). The meaning of the first loop is given by the statement

[3] In Alphard, certain functions are given names beginning with "&". These are usually functions provided by the user to perform operations that correspond to special constructs of the language. Outside the form in which they are defined, they may *not* be called by user programs. In this case, the for loop expects to call functions named &init and &next with certain specified properties. Alphard prevents a user from calling them explicitly -- to skip iterations in a loop, for example.

[4] Either "then S1" or "else S2" may be omitted; an omitted clause is assumed to denote the empty statement.

```
begin label λ;
    begin local x: gen(y);
        π ← x.&init;
        while π do
            if β(x) then (S1(x,y,z); goto λ) else π ← x.&next
    end;
  S2(y,z);
λ: end
```

As above, the compiler-generated names, π and λ, are not accessible to the programmer.

In [Wulf76a,b] we presented a subroutine to compare two vectors of arbitrary (but identical) types and index sets. The subroutine presented there was phrased in terms of an Algol-like for loop. It can now be written in real Alphard using the first statement:[5]

```
function eqvecs(A,B: vector(?t<≠>,?lb,?ub)) returns (eq: boolean) =
   first i: upto(lb,ub) suchthat A[i] ≠ B[i] then eq ← false else eq ← true
```

It does not matter what the bounds of the two vectors are, as long as they are the same. In this case, we are not relying on the procedure return or an explicit escape to terminate the loop early in the case of inequality; that is handled by the first statement. The proof of "eqvecs" will be given in a later section.

We have introduced Alphard loop constructs by comparing them to simple counting loops. This is the first step toward solving the problem of sequencing over arbitrary structures under the control of the defining type. We shall now show how generators and loops are verified.

Defining and Verifying Generators

We said that a generator is a form which supplies special functions and performs a sequence of bindings to the control variable of the loop. In this section we will show how a generator is defined and invoked, still using "upto" as an example. We will first present its definition, then add assertions, verify it as a form, and establish its special properties as a generator. Another generator is verified as part of the finite sets example in the sequel.

---

[5] In this example the function specification and the function body are given as one declaration. This is an obvious abbreviation of the notation used elsewhere. The *?identifier* notation is used to indicate that the values of these parameters must be identical for A and B and that specific values will be supplied implicitly with the vectors. This is explained in [Wulf76a,b].

The definition of the "upto" generator, without verification information, is

```
form upto(lb,ub: integer) extends k:integer =
   beginform
   specifications
       inherits <allbut ←>;
       function
           &init(u:upto) returns (b:boolean),
           &next(u:upto) returns (b:boolean);
   implementation
       body &init = (u.k ← u.lb; b ← u.lb ≤ u.ub);
       body &next = (u.k ← u.k+1; b ← u.k ≤ u.ub);
   endform
```

Since no variables other than k are needed, the representation part is empty at this point. This form extends integers, but does not pass along the right to assign to an upto;[6] this prevents the user from changing the loop variable during the iteration.

Using this form and the meaning of the for statement given in the previous section, we can exhibit a loop that corresponds to the expansion of the "upto" functions in the statement for summing integers. This code is, of course, only suggestive, but it illustrates an expansion which a compiler might reasonably produce. Note that an obvious optimization has been applied; later, when we exhibit the formal specifications of "upto", the value of the iteration variable, x, will turn out to be irrelevant when &init or &next returns false.

```
sum ← 0;
begin
   local x: upto(lb,ub);
       x ← x.lb;
       while x≤x.ub do (sum←sum+x; x←x+1);
end
```

Since "upto" is a form, we can verify the form properties as described in [Wulf76a,b] and summarized in Appendix A. Adding verification information in italics, the definition of "upto" becomes

6 The phrase "allbut ←" means that all integer functions *except* ← are applicable to the upto.

```
form upto(lb,ub: integer) extends k: integer =
   beginform
   specifications
       requires true;
       inherits <allbut ←>;
       let upto = [lb..ub] where lb ≤ ub ⊃ upto = [lb..k-1][k][k+1..ub];
       invariant true;
       initially true;
       function
           &init(u:upto) returns (b:boolean)
               post (b ≡ lb≤ub) ∧ (b ⊃ lb=k≤ub),
           &next(u:upto) returns (b:boolean)
               pre lb ≤ k ≤ ub
               post (b ≡ k'<ub) ∧ (b ⊃ k=k'+1 ∧ lb≤k≤ub);
   representation
       rep(k) = if lb≤ub then [lb..k-1][k][k+1..ub] else [];
       invariant true;
   implementation
       body &init out (b ≡ lb≤ub) ∧ (b ⊃ lb=k≤ub) =
           (u.k ← u.lb; b ← u.lb ≤ u.ub);
       body &next in lb≤k≤ub out (b ≡ k'<ub) ∧ (b ⊃ k = k'+1 ∧ lb≤k≤ub) =
           (u.k ← u.k+1; b ← u.k ≤ u.ub);
   endform
```

The abstract specifications describe an "upto" as an interval [lb..ub]; since the form upto extends the integer k, a direct reference to a loop variable of type upto will access k, the current value of the loop counter. We will find it useful later to view the upto as the concatenation of the interval already processed ([lb..k-1]), the current element ([k]), and the interval yet to be generated ([k+1..ub]). Either k stays between the endpoints of the interval [lb..ub] or the interval is empty. This is enforced by the phrase lb≤k≤ub which appears in the pre condition for &next and both post conditions.

Note that no promise about the value of k is made before the loop starts (i.e., before &init is called) or after it has run to completion (either &init or &next returns false). The rep function shows how an interval is represented by its two endpoints and the loop variable. The post condition on &init guarantees that the first element generated is lb, but only if lb≤ub. The pre condition on &next prevents &next from being executed when there is no valid current element (in particular, &init must be called first). The post condition on &next guarantees that generated values are consecutive and that the generator stops at ub.

For "upto" the four steps which are required to verify the form properties are quite simple. (Note that the "u." qualification on u.lb, u.k, and u.ub is omitted for simplicity.)

For the form

1. Representation validity
   Show: true ⊃ true
   Proof: clear

2. Initialization
   Show: true { } true ∧ true
   Proof: clear

For the function &init

3. Concrete operation
   Show: true { k ← lb; b ← lb ≤ ub } (b≡lb≤ub) ∧ (b⊃lb=k≤ub)
   Proof: Using the assignment axiom, the expression becomes
   true ⊃ (lb≤ub ≡ lb≤ub) ∧ (lb≤ub ⊃ lb=lb≤ub)
   which surely holds.
4. Relation Between Abstract and Concrete
   Corresponding abstract and concrete assertions are identical and the rep function performs a direct mapping, so the proofs are clear.

For the function &next

3. Concrete operation
   Show: lb≤k≤ub { k ← k+1; b ← k≤ub } (b≡k'<ub) ∧ (b⊃k=k'+1 ∧ lb≤k≤ub)
   Proof: Using the assignment axiom, the expression becomes
   lb≤k≤ub ⊃ (k+1≤ub ≡ k'<ub) ∧ (k+1≤ub ⊃ k+1=k'+1 ∧ lb≤k+1≤ub)
   which holds because k'=k is an implicit hypothesis of the antecedent.
4. Relation Between Abstract and Concrete
   Same as &init.4.

QED

To emphasize that a generator is a form, we will now give an example in which a generator is instantiated in one place and used in another. The following procedure is a generalized sum routine. Its parameter is an instantiation of a generator and its result is the sum of the elements produced by that generator. For simplicity, this procedure sums only integers. That restriction can be relaxed, but to do so would take us into parts of Alphard not discussed in this paper.[7]

---

[7] The difficulty is not defining the type of the output, which would be expressed as

function ISUM (g: ?T<generator extends ?S>) returns (sum: S)

but rather the fact that we need to initialize sum and do not know the identity for "+" in type S. One solution is to treat the first generated element differently from the rest, and we have deferred discussion of the richer possibilities of generators to a later paper.

Definition

```
function ISUM (g: ?T<generator extends integer>) returns (sum: integer)
    begin
    sum ← 0;
    for g do sum ← sum + g;
    end
```

Examples of Use

```
begin
local v: vector(integer,1,n),
    ig: upto(1,m), vg: invec(v),
    ssum, vsum: integer;
. . .
ssum ← ISUM (ig);
vsum ← ISUM (vg);
end
```

This small program declares five variables. The first, v, is a vector of integers indexed from 1 to n. The next two, vg and ig, are (instantiations of) generators; ig is an instance of the upto we have been discussing and vg is an invec, which we assume is defined along with vectors and generates the elements of the vector named as its instantiation parameter. The last two variables, ssum and vsum, are simple integers. The first call on ISUM uses ig (the upto) to generate integer values; it assigns to ssum the sum of the integers from 1 to m. The second call on ISUM uses vg (the invec) to generate vector elements; it assigns to ssum the sum of the elements of v.

Proof Rules for Loops

In this section we shall consider the verification of Alphard's two iteration constructs, <u>for</u> and <u>first</u>. Specifically, we shall develop proof rules for these statements, discovering in the process certain desirable properties for <u>forms</u> which are intended to be used as generators. Some of these properties will be required of all generators; others will be considered optional, but their presence will substantially simplify proof rules and proofs.

The development will proceed as follows. First we shall consider a proof rule for the <u>for</u> statement which makes minimal assumptions about the generator. This rule is derived directly from the statement's meaning as given earlier. As a consequence, it is rather bulky. Then we shall make a small number of basic assumptions about the generator. For purposes of this paper, these assumptions will be required of all generators and hence will have to be discharged when the generator is verified as a <u>form</u>. They will allow us to simplify substantially the proof rules for the <u>for</u> and <u>first</u> statements. Next we shall consider a further set of assumptions about generators; these assumptions are not mandatory, but they are satisfied by typical generators. These will allow us to obtain still simpler proof rules for particular generators. Finally, we shall consider the properties that a generator must have in order to be a *terminating generator*.

Development of the for Rule

Suppose that we wish to prove

$$P \{ \underline{for}\ x{:}gen(y)\ \underline{while}\ \beta(x)\ \underline{do}\ ST(x,y,z) \mid I(x,y,z) \} Q$$

where x, y, and z are as defined earlier and the notation "P { loop | I } Q" is used to denote "P { loop } Q using I as the loop assertion (invariant) placed *after* the loop body". Further, suppose that we make only the minimal assumptions about the form "gen", namely that it has been verified as a form and that it supplies two functions, &init and &next, each of which takes a single parameter of type gen and returns a boolean result. We will also assume that $\beta(x)$ has no side effects. We will adopt the following notation in the iteration proof rules:

G = abstract invariant of the generator. G may depend on x and y but not on z.

β_{req} = the usual requires clause of the generator, stating restrictions on y so that the generator can be instantiated.[8]

$\beta_{f.j}$ = the j-condition for generator function f, e.g., $\beta_{init.post}$ is the post condition for &init. $\beta_{f.j}$ depends on x and y only.

$x_0,\dots,x_p$ denotes the previously generated values of x, if any.

Since the generator has been verified as a form, we know

$$G \wedge \beta_{init.pre} \{ \pi \leftarrow x.\&init \} G \wedge \beta_{init.post}$$
$$G \wedge \beta_{next.pre} \{ \pi \leftarrow x.\&next \} G \wedge \beta_{next.post}$$
$$\beta_{req} \{ \textit{init clause} \} G$$

where *init clause* denotes the init clause of the representation part.

The expansion of

$$\underline{for}\ x{:}gen(y)\ \underline{while}\ \beta(x)\ \underline{do}\ ST(x,y,z)$$

---

8 We conventionally use "β" to name predicates. Hence, e.g., β_{req} is unrelated to $\beta(x)$.

as a standard while statement, including the assertions which will be required for verification in the most general case, is

```
assert P ∧ β_req;
begin local x: gen(y);
assert P ∧ G ∧ β_init.pre;
π ← x.&init;
while π cand β(x) do
   begin
      ST(x,y,z);
      assert I ∧ G ∧ β_next.pre;
      π ← x.&next;
   end;
end;
assert Q
```

We will give from this expansion a proof rule for the most general Alphard for statement. The standard while rule is not directly applicable to this expansion because the loop-cutting assertion is located in the middle of the loop body rather than before the test. This assertion placement means the test does not always appear just before or just after an assertion; in two control paths through the expansion (the third and fifth lines in the proof rule below), the test π *cand* β(x) appears between either the statements π←x.&init or π←x.&next and ST(x,y,z). To indicate in these paths that π *cand* β(x) may be assumed between the statements, the assume clause is introduced.[9] Its proof rule is

$$\frac{P \wedge Q \supset R}{P \; \{ \; \underline{\text{assume}} \; Q \; \} \; R}$$

Using the assume clause and considering the five control paths between assertions, the general proof rule for the for statement is

$$\frac{\begin{array}{l} P \wedge \beta_{req} \; \{ \; \textit{init clause} \; \} \; P \wedge \beta_{init.pre} \\ P \wedge G \wedge \beta_{init.pre} \; \{ \; \pi \leftarrow x.\&init \; \} \; \neg(\pi \wedge \beta(x)) \supset Q \\ P \wedge G \wedge \beta_{init.pre} \; \{ \; \pi \leftarrow x.\&init; \; \underline{\text{assume}} \; \pi \wedge \beta(x); \; ST(x,y,z) \; \} \; I \wedge G \wedge \beta_{next.pre} \\ I \wedge G \wedge \beta_{next.pre} \; \{ \; \pi \leftarrow x.\&next \; \} \; \neg(\pi \wedge \beta(x)) \supset Q \\ I \wedge G \wedge \beta_{next.pre} \; \{ \; \pi \leftarrow x.\&next; \; \underline{\text{assume}} \; \pi \wedge \beta(x); \; ST(x,y,z) \; \} \; I \wedge G \wedge \beta_{next.pre} \end{array}}{P \wedge \beta_{req} \; \{ \; \underline{\text{for}} \; x: gen(y) \; \underline{\text{while}} \; \beta(x) \; \underline{\text{do}} \; ST(x,y,z) \mid I \; \} \; Q}$$

---

9 The assume clause appears in [Igarashi75, p. 164] as the "marked" assertion using the notation Q-if in place of assume Q.

This formulation, because of its generality, may appear formidable. The main difficulty appears to be that the three generator functions and the loop body may each change y in various ways even though P and I hold at the places required by the rule. The generator functions are, therefore, involved in the verification of each use of a generator. However, the following three reasonable assumptions about the generator will simplify matters considerably.

Basic Generator Assumptions:

(a) The post conditions on &init and &next are of the form

$$(b \equiv \pi_i) \wedge \beta_i \text{ and } (b \equiv \pi_n) \wedge \beta_n$$

respectively, where b is the result parameter of these functions.

(b) $G \supset \beta_{init.pre}$, $G \wedge (\pi_i \wedge \beta_{init.post} \vee \pi_n \wedge \beta_{next.post}) \supset \beta_{next.pre}$

(c) The *init clause* and the functions &init and &next terminate. (This does not simplify the proof rule. It is, however, a desirable property, and it becomes especially relevant in the discussion of generator termination below.)

(d) The generator and the loop body are *independent*. That is, for arbitrary predicates R and S

R(y,z) { *init clause* } R(y,z)
R(y,z) { $\pi \leftarrow$ x.&init } R(y,z)
R(y,z) { $\pi \leftarrow$ x.&next } R(y,z)
and S(x,y) { ST(x,y,z) } S(x,y)

Point (a) is a minor restriction and can be checked syntactically. Point (b) requires two proofs. The first is usually trivial since $\beta_{init.pre}$ is generally omitted (defaulted to true) and $\beta_{next.pre}$ is usually included in both post conditions. G may often be strong enough by itself, but we may not want to commit the generator to provide a value at all times. In the latter case we therefore require that &init and &next make it possible for &next to be executed. Point (c) can be proved independently of the use of the generator. The proofs should usually be easy (see the section below on termination).

Point (d) requires four proofs; in the typical case, however, the first three are trivial. Because of the scope restrictions mentioned in [Wulf76a,b], the only ways the *init clause*, &init or &next could affect the predicate R(y,z) are through y, which is explicitly passed as a parameter to the form gen, and through side-effect-producing operations of &init and &next. Thus the proof can be carried out locally for the generator definition -- generally by inspection. The fourth proof is more difficult. Because of the scope restrictions, the only way that the loop body could affect the loop variable, x, is for the generator to provide a function which could have a side effect on x (for example, by exporting assignment rights). This proof should be local to the generator definition. However, the independence of y from ST cannot in general be shown for the generator, and must be treated as a restriction on its use.

Simplified Rules for Iteration Statements

If the generator and its use meet the four basic generator assumptions given above, a simplified proof rule applies to the <u>for</u> statement:[10]

$$G \wedge [P \wedge \beta_i \wedge \neg(\pi_i \wedge \beta(x)) \vee I \wedge \beta_n \wedge \neg(\pi_n \wedge \beta(x))] \supset Q$$
$$G \wedge \beta(x) \wedge [P \wedge \beta_i \wedge \pi_i \vee I \wedge \beta_n \wedge \pi_n] \{ ST(x,y,z) \} I$$

$$P \wedge \beta_{req} \{ \underline{for}\ x: gen(y)\ \underline{while}\ \beta(x)\ \underline{do}\ ST(x,y,z) \mid I \} Q$$

Note that the first line establishes that Q holds when (if) the loop terminates -- which may happen immediately after the invocation of &init (handled by the first term of the disjunction in []'s), or after an invocation of &next (handled by the second term of the disjunction). In both cases termination may result either because the relevant generator function returned false or because $\beta(x)$ failed -- hence the terms of the form "$\neg(\pi \wedge \beta(x))$". The second line ensures that the invariant is established after each application of the loop body.

Under the same assumptions, the following proof rule applies to the <u>first</u> statement:

$$G \wedge P \wedge [\beta_i \wedge \pi_i \vee \beta_n \wedge \pi_n \wedge \neg\beta(x_0..x_p)] \wedge \beta(x) \{ S_1(x,y,z) \} Q$$
$$G \wedge P \wedge [\neg\pi_i \wedge \beta_i \vee \neg\pi_n \wedge \beta_n \wedge \neg\beta(x_0..x_p)] \{ S_2(y,z) \} Q$$

$$P \wedge \beta_{req} \{ \underline{first}\ x: gen(y)\ \underline{suchthat}\ \beta(x)\ \underline{then}\ S_1(x,y,z)\ \underline{else}\ S_2(y,z) \} Q$$

where "$\neg\beta(x_0..x_p)$" is an abbreviation for "$\neg\beta(x_0) \wedge \ldots \wedge \neg\beta(x_p)$". Note that the second line handles the "else" cases, where no match is found; the two terms of the disjunction are the case where the generator terminates immediately and the case where every element generated fails the <u>suchthat</u> test, $\beta(x)$. The first line handles the case where a match is found. Note also that the presumed independence of the generator and the user program means that P is not affected by &init and &next.

Simplified Rules for Typical Generators

Most generators are far more stylized than the simple assumptions above require. The

---

[10] The justifications of this and the <u>first</u> rule, from the corresponding general rules and the basic generator assumptions, are given in Appendix B.

following assumptions about standard aggregates used in typical generators allow us to obtain proof rules of further simplicity.

Standard Aggregate Assumptions

(a) The additional abstraction provided by the generator is explicated in terms of an aggregate (of objects of the base type) for which the following are defined:

@	an operator to combine (e.g., concatenate) two aggregates
<>	the empty aggregate
lead(S) =	first element of S to be generated.

Examples of such aggregates are sets, sequences, and intervals. The corresponding empty aggregates are {}, <>, and []; the corresponding @ operators are union, concatenation, and merging adjacent intervals.

(b) The instantiation of the generator will produce the complete aggregate, T, of objects to be generated. Further, a nonempty T can be decomposed as

$$T = s \mathbin{@} \langle x \rangle \mathbin{@} t$$

where: <x> is the unit aggregate consisting of the current element x; s and t are (possibly empty) aggregates -- s, those elements previously generated and t, those remaining to be generated; and s, <x>, and t are mutually disjoint.

(c) The specifications on &init and &next have the form

```
functions
    &init(&g:gen) returns &b:boolean
        post (&b ≡ T≠<>) ∧ (&b ⊃ x=lead(T) ∧ D1(x))
    &next(&g:gen) returns &b:boolean
        pre D2(x)
        post (&b ≡ t'≠<>) ∧ (&b ⊃ x=lead(t') ∧ D3(x))
```

where &g is an instantiation of gen corresponding to the aggregate T and the $D_i(x)$ guarantee that the decomposition of T specified in (b) is legal and can be found.

The standard aggregate assumptions subsume points (a) and (b) of the basic generator assumptions, but points (c) and (d) of the latter must still be demonstrated in addition to the standard aggregate assumptions.

If these assumptions hold, we can derive several simpler proof rules. The rule for the <u>for</u> statement becomes

$$\frac{\begin{array}{c} G \wedge [P \wedge (T=\langle\rangle \vee \neg\beta(\mathrm{lead}(T))) \vee T\neq\langle\rangle \wedge I(s) \wedge (s=T \vee \neg\beta(x))] \supset Q \\ G \wedge T\neq\langle\rangle \wedge [P \wedge \beta(\mathrm{lead}(T)) \vee (s\neq T \wedge I(s) \wedge \beta(x))]\ \{\ ST\ \}\ I(s@\langle x\rangle) \end{array}}{P \wedge \beta_{req}\ \{\ \underline{\text{for}}\ x\text{: gen}(y)\ \underline{\text{while}}\ \beta(x)\ \underline{\text{do}}\ ST(x,y,z) \mid I\ \}\ Q}$$

and the <u>first</u> rule simplifies to

$$\frac{\begin{array}{c} G \wedge P \wedge \forall w \in s\ \neg\beta(w) \wedge \beta(x)\ \{\ S_1(x,y,z)\ \}\ Q \\ G \wedge P \wedge \forall w \in T\ \neg\beta(w)\ \{\ S_2(y,z)\ \}\ Q \end{array}}{P \wedge \beta_{req}\ \{\ \underline{\text{first}}\ x\text{: gen}(y)\ \underline{\text{suchthat}}\ \beta(x)\ \underline{\text{then}}\ S_1(x,y,z)\ \underline{\text{else}}\ S_2(y,z)\ \}\ Q}$$

We call these two rules the standard aggregate rules.

Special Cases and Examples

The Pure <u>for</u> Rule

In many cases the programmer may wish to drop the <u>while</u> clause, treating $\beta(x)$ as identically true. In addition, he will often wish to choose $P = I(\langle\rangle)$ and $Q = I(T)$. (Until now the major reason for distinguishing between P, Q, and I was that if $\beta(x)$ terminates the loop before the generator signals termination, $I(T)$ is probably not true.) If these decisions are made, the proof rule simplifies further, since the first premise reduces to true and several terms drop out of the second. Making the substitutions yields a generic rule similar to those of various <u>for</u> statements given by Hoare [Hoare72a]:

$$\frac{G \wedge T=s@\langle x\rangle@t \wedge I(s)\ \{\ ST(x,y,z)\ \}\ I(s@\langle x\rangle)}{I(\langle\rangle) \wedge \beta_{req}\ \{\ \underline{\text{for}}\ x\text{: gen}(y)\ \underline{\text{do}}\ ST(x,y,z)\ \}\ I(T)}$$

Proof Rules for upto

To use one of these rules with a particular generator, we must "instantiate" it with the particulars of the generator in question. We will illustrate this by developing the proof rules for upto. First, we discharge parts (c) and (d) of the basic generator assumptions:

(c) The bodies consist of simple assignment statements, and thus clearly terminate.

(d) There is no init clause and functions &init and &next change only local data and their return values; thus the first three parts of independence are satisfied. For the fourth point, note that no means is provided for the user of the <u>form</u> to alter k; the user is expected to refrain from altering lb and ub.

Next, we discharge the standard aggregate assumptions:

(a) Integer intervals are used.

(b) $[lb..ub] = [lb..k-1][k][k+1..ub]$ when $lb \le k \le ub$.

(c) The <u>pre</u> and <u>post</u> conditions have the required form.

Substituting the interval definitions in the standard aggregate rules and simplifying, we obtain

$$\frac{\begin{array}{c} P \land (lb>ub \lor \neg\beta(lb)) \lor lb\le k\le ub \land I[lb..k-1] \land \neg\beta(k) \lor lb\le ub \land I[lb..ub] \supset Q \\ lb\le ub \land (P \land \beta(lb) \lor lb\le k\le ub \land I[lb..k-1] \land \beta(k))\ \{\ ST(k,y,z)\ \}\ I[lb..k] \end{array}}{P\ \{\ \underline{\text{for}}\ k\text{: upto}(lb,ub)\ \underline{\text{while}}\ \beta(k)\ \underline{\text{do}}\ ST(k,y,z)\ |\ I(k,y,z)\ \}\ Q}$$

and

$$\frac{\begin{array}{c} P \land lb\le k\le ub \land (\forall w \in [lb..k-1]\ \neg\beta(w)) \land \beta(k)\ \{\ S_1(k,y,z)\ \}\ Q \\ P \land \forall w \in [lb..ub]\ \neg\beta(w)\ \{\ S_2(y,z)\ \}\ Q \end{array}}{P\ \{\ \underline{\text{first}}\ k\text{: upto}(lb,ub)\ \underline{\text{suchthat}}\ \beta(k)\ \underline{\text{then}}\ S_1(k,y,z)\ \underline{\text{else}}\ S_2(y,z)\ \}\ Q}$$

where the y parameters are <lb,ub>. In the special case $P=I[]$, $Q=I[lb..ub]$, and $\beta\equiv$true, we obtain the Pascal rule for the <u>for</u> statement [Hoare72a, Hoare73]:

$$\frac{lb\le k\le ub \land\ I[lb..k-1]\ \{ST(k,y,z)\}\ I[lb..k]}{I[]\ \{\ \underline{\text{for}}\ k\text{:upto}(lb,ub)\ \underline{\text{do}}\ ST(k,y,z)\ \}\ I[lb..ub]}$$

As must be the case, this rule is also obtained from the pure <u>for</u> rule by instantiating gen(y) with upto(lb,ub).

The Pure while Rule

We showed above that when the while clause is dropped, the for proof rule resembles Hoare's. We will now show how to eliminate the loop variable and obtain the standard proof rule for the pure while statement.

Suppose we had a form named "forever" which extended type boolean and which satisfied the requirements above by using the value "true" for all the predicates involved. The aggregate T would be an infinite sequence of "true"s, and the standard aggregate for rule would become

$$\frac{\begin{array}{c}\text{true} \wedge [P \wedge (\text{false} \vee \neg\beta(\text{true})) \vee \text{true} \wedge I(\text{true}^*) \wedge (\text{false} \vee \neg\beta(\text{true}))] \supset Q \\ \text{true} \wedge [P \wedge \beta(\text{true}) \vee \text{true} \wedge I(\text{true}^*) \wedge \beta(\text{true})]\ \{\ ST(\text{true},,z)\ \}\ I(\text{true}^*)\end{array}}{P\ \{\ \underline{\text{for}}\ x\text{: forever}\ \underline{\text{while}}\ \beta(\text{true})\ \underline{\text{do}}\ ST(\text{true},,z) \mid I(\text{true}^*)\ \}\ Q}$$

where "true*" denotes a sequence of "true"s and the adjacent commas indicate the absence of the parameters y. By choosing P = I and Q = I ∧ ¬β, eliminating the vacuous dependencies on "true", dropping the useless for clause, and simplifying, we obtain

$$\frac{I \wedge \beta\ \{\ ST(z)\ \}\ I}{I\ \{\ \underline{\text{while}}\ \beta\ \underline{\text{do}}\ ST(z)\ \}\ I \wedge \neg\beta}$$

which is the conventional while rule.

Generator Specifications by Proof Rules

We have shown how two sets of assumptions about the properties of a generator lead to very simple proof rules for the iteration statements. Notice now that if a generator satisfies these assumptions, the specifications for &init and &next can be *reconstructed* or *obtained* from the proof rules. As a result, the author of the generator can perform the substitutions and simplifications, then give the proof rules in the specifications instead of giving the pre and post conditions. When this is possible, we use the keyword generator in place of form in the specification to alert the user.

To illustrate this, we will write the generator for a counting loop that uses an integer step size greater than 1. This will provide the Alphard equivalent of Algol's

for i := a step j until b do S

for positive values of j. We first augment the interval notation [a..b] to include a step size:

$$[a(j)b] \equiv_{df} \langle a,a+j,a+2*j, \ldots ,b-(b-a) \bmod j\rangle \ \underline{\text{where}}\ j>0$$

If a>b, then [a(j)b] is <>. Note that [a(1)b] = [a..b]. The following rule allows us to merge two intervals:

$$[a(j)b][b+j(j)c]=[a(j)c] \quad \text{provided} \quad (b-a) \bmod j = 0$$

Using this notation, we can define the generator *stepup*:

```
generator stepup (lb,j,ub:integer)extends k:integer=
   beginform
   specifications
       requires j > 0;
       inherits <allbut ←>;
       let stepup = [lb(j)ub] where lb≤ub ⊃ stepup = [lb(j)k-j][k][k+j(j)ub];

       rule forwhile(P∧j>0, k, <lb,j,ub>, β, ST(k,<lb,j,ub>,z), I, Q) =
           premise P∧(lb>ub∨¬β(lb)) ∨ lb≤k≤ub-d∧I[lb(j)k-j]∧¬β(k) ∨ lb≤ub∧I[lb(j)ub] ⊃ Q,
           premise lb≤ub ∧ (P∧β(lb) ∨ lb≤k≤ub-d∧I[lb(j)k-j]∧β(k)) { ST(k,<lb,j,ub>,z) }
                       I[lb(j)k] where d=(ub-lb) mod j;

       rule first(P∧j>0, k, <lb,j,ub>, β, S1(k,<lb,j,ub>,z), S2(<lb,j,ub>,z), Q) =
           premise P ∧ lb≤k≤ub ∧ (∀w ∈ [lb(j)k-j] ¬β(w)) ∧ β(k) { S1(k,<lb,j,ub>,z) } Q,
           premise P ∧ ∀w ∈ [lb(j)ub] ¬β(w) { S2(<lb,j,ub>,z) } Q;

       rule for(I∧j>0, k, <lb,j,ub>, ST(k, <lb,j,ub>, z)) =
           premise lb≤k≤ub-d ∧  I[lb(j)k-j] { ST(k,<lb,j,ub>,z) } I[lb(j)k]
                       where d=(ub-lb) mod j;
   representation
       !
       ! same as upto
       !
   implementation
       !
       ! same as upto, except in &next "+1" becomes "+j" and k'<ub becomes k'+j≤ub
       !
   endform
```

Example of Loop Verification

In this section we shall illustrate the use of the proof rules given above by verifying the "eqvecs" function given earlier. With <u>pre</u> and <u>post</u> assertions, the function is

```
function eqvecs(A,B: vector(?t<≠>,?lb,?ub)) returns (eq: boolean) =
   pre true post (eq ≡ (∀j ∈ [lb..ub] A[j]=B[j])) =
   first i: upto(lb,ub) suchthat A[i] ≠ B[i] then eq ← false else eq ← true
```

Using the upto first rule, the proof requires that we establish the two premises:

Show: $\text{true} \wedge lb \le i \le ub \wedge (\forall w \in [lb..i-1]\ \neg(A[y] \ne B[y])) \wedge A[i] \ne B[i]$
$\{\ eq \leftarrow \text{false}\ \}\ eq \equiv \forall j \in [lb..ub]\ A[j]=B[j]$

Proof: This simplifies to $lb \le i \le ub \wedge A[i] \ne B[i] \supset \exists j \in [lb..ub]\ A[j] \ne B[j]$. Choose $j=i$.

Show: $\text{true} \wedge \forall w \in [lb..ub]\ \neg(A[w] \ne B[w])\ \{\ eq \leftarrow \text{true}\ \}\ eq \equiv \forall j \in [lb..ub]\ A[j]=B[j]$

Proof: clear

QED

Termination of Generators

A major advantage of the for statements in many of the more recent programming languages, such as Pascal, is that they are guaranteed to terminate (provided, of course, that the statement which is the loop body terminates for each value of the for statement). As a result the programmer using them never need explicitly demonstrate termination. We would like to be able to make similar claims about the loops utilizing at least some generators; the generators having this property will be called *terminating generators.*

We can now present a technique for demonstrating this property.[11] Although the general for statement is

for x:gen(y) while β(x) do ST(x,y,z)

the clause "while β(x)" can only reduce the number of times ST(x,y,z) is executed. Hence it suffices to show that

for x:gen(y) do ST(x,y,z)

terminates. Further, the generator and loop body, ST(x,y,z), are independent, so we know that as long as the body itself terminates for each x, it cannot cause the for statement to fail to

[11] Note that nontermination of the loop might also be caused by nonterminaton of the *init clause* or the functions &init and &next in the generator. This is explicitly ruled out by the basic generator assumptions, but must be treated as an additional requirement for proof of termination of generators which do not satisfy those assumptions.

terminate. Thus, if we can show the termination of the above statement for all possible parameters of the generator and some *particular* loop body, we will have shown that use of the generator cannot cause nontermination for any body.

Consider the statement

$$i \leftarrow 0;\ \underline{\text{for}}\ x{:}gen(y)\ \underline{\text{do}}\ i \leftarrow i+1$$

If we could find: (1) a (non-negative) value M_y depending only on y for which $i \le M_y$ after executing the statement, and (2) a loop invariant which allowed us to prove that the loop terminated with such a value of i, then we would have proved termination of all loops using gen.

Clearly, the choice of M_y will depend on the instantiation parameters of the generator, i.e., on the data structure from which the elements are being generated. The loop invariant will have to assert that M_y bounds i; it will also have to relate the value of i to progress through the loop. The term that accomplishes the latter task, which we shall call $I_y(x)$, must be chosen for each generator whose termination is to be proved. Thus the loop invariant is of the form $i \le M_y \wedge I_y(x)$. If we can associate with a generator a rule for determining M_y for any particular instantiation, and if we can find a suitable $I_y(x)$, then it suffices to show[12]

$$i=0\ \ \{\ \underline{\text{for}}\ x{:}gen(y)\ \underline{\text{do}}\ i \leftarrow i+1 \mid i \le M_y \wedge I_y(x)\ \}\ i \le M_y$$

Note that the clause "$i \le M_y$" in this loop invariant ensures that the loop will terminate, since i is strictly increasing from 0.

Although this must potentially be proved for each generator, we can show the termination of every generator which satisfies the standard aggregate assumptions (with a finite aggregate), provided only that it is possible to measure the size of an aggregate. To demonstrate this, we use the pure for rule taking I(s) as $i \le size(T) \wedge i=size(s)$, where "size" is defined appropriately for the aggregate. The only premise

$$G \wedge T=s@\langle x\rangle@t \wedge i \le size(T) \wedge i=size(s)\ \{\ i \leftarrow i+1\ \}\ i \le size(T) \wedge i=size(s@\langle x\rangle)$$

follows since s and <x> are disjoint, whence size(s) < size(T) and size(s@<x>) = size(s)+1. Hence the conclusion of the pure for rule is

$$i \le size(T) \wedge i=size(\langle\rangle)\ \{\ \underline{\text{for}}\ x{:}\ gen(y)\ \underline{\text{do}}\ i \leftarrow i+1\ \}\ i \le size(T) \wedge i=size(T)$$

This then implies the desired result with M_y=size(T) and $I_y(x)$=size(s).

---

[12] This method for showing termination is a simple instance of the commonly-used well-founded set notion [Katz75, Luckham75]. Here the well-founded set is the non-negative integers bounded by M_y.

Example: Finite Sets

We now turn to a larger example that uses the iteration constructs. This example is based on Hoare's "smallintset" [Hoare72b], which implements small sets of integers. We begin by presenting and verifying a slightly augmented version of "smallintset". This form, called "simpleset", uses first statements and the "upto" generator; the program and the verification can be compared with Hoare's "smallintset". We then discuss the problem of adding new operations to "simpleset"; we construct a new type with the additional operators by adding a set-element generator to "simpleset" and writing a new form (which extends "simpleset") for the new operators.

"Simpleset": a Version of Hoare's "Smallintset"

This differs from Hoare's "smallintset" in that it can build sets of many types and the bound on the set size can be selected for each instantiation. Hoare noted these extensions in [Hoare72b, section 9]. In addition, the algorithm used in "remove" is slightly different.[13]

```
form simpleset(maxsize:integer, thing:form<←,=>) =
   beginform
   specifications
       requires maxsize ≥ 0;
       let simpleset = { . . . x_i . . . } where x_i is thing;
       invariant cardinality(simpleset) ≤ maxsize;
       initially simpleset = {};
       function
           insert(s:simpleset, x:thing)
               pre cardinality({x} ∪ s) ≤ maxsize
               post s = s' ∪ {x},
           remove(s:simpleset, x:thing)
               post s = s' - {x},
           has(s:simpleset, x:thing) returns (b: boolean)
               post b = x ∈ s';
```

[13] 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 has below. Such omitted assertions are nevertheless used in the proof steps.

```
representation
    unique v: vector(thing,1,maxsize),  m: integer init m ← 0;
    rep(v,m) = {v[i] | i ∈ [1..m]};
    invariant 0≤m≤maxsize ∧ (∀i,j ∈ [1..m] (v[i]=v[j] ⊃ i=j));

implementation
    body insert in (∃i ∈ [1..s.m] st x=s.v[i] ∨ s.m<maxsize)
            out (∀i∈[1..s.m'](s.v[i] = s.v'[i]) ∧ (∃j ∈ [1..s.m] st s.v[j] = x)) =
        first p: upto(1,s.m) suchthat s.v[p] = x
            else (s.m ← s.m+1; s.v[s.m] ← x);

    body remove out (∀j ∈ [1..s.m](s.v[j] ≠ x) ∧
            (∀i ∈ [1..s.m'] ∃j ∈ [1..s.m] (s.v'[i] ≠ x ⊃ s.v[j] = s.v'[i])))=
        first p: upto(1,s.m) suchthat s.v[p] = x
            then (s.v[p] ← s.v[s.m]; s.m ← s.m-1);

    body has out (b ≡ (∃i ∈ [1..s.m] st s.v[i]=x) ∧ s.v'=s.v ∧ s.m'=s.m) =
        first p: upto(1,s.m) suchthat s.v[p] = x
            then b ← true else b ← false;

endform
```

Verification of Simpleset

For the form

1. Representation validity
   Show: 0≤m≤maxsize ∧ (∀i,j ∈ [1..m](v[i]=v[j]⊃i=j)) ⊃
   cardinality({v[i] | i ∈ [1..m]})≤maxsize)
   Proof: clear

2. Initialization
   Show: maxsize≥0 {m←0} {v[i] | i ∈ [1..m]}={} ∧ 0≤m≤maxsize ∧
   ∀i,j∈[1..m](v[i]=v[j] ⊃ i=j))
   Proof: 0≤0≤maxsize and [1..0] is [].

For the function insert

3. Concrete operation
   Show: $\beta_{in} \wedge I_c$ { first p: upto(1,s.m) suchthat s.v[p]=x
   else (s.m←s.m+1; s.v[s.m]←x } $\beta_{out} \wedge I_c$
   Proof: The second premise of the upto first rule becomes

$(\exists i \in [1..s.m]$ *st* $x=s.v[i] \lor s.m<maxsize) \land I_c \land$
$\forall k \in [1..s.m](s.v[k]\neq x)$ { s.m←s.m+1; s.v[s.m]←x }
$\forall i \in [1..s.m'](s.v[i]=s.v'[i]) \land (\exists j \in [1..s.m]$ *st* $s.v[j]=x) \land I_c$

The first term follows by s.m=s.m'+1>s.m'. For the second term choose j=s.m (note 1≤s.m≤maxsize). The first term of I_c holds because the ∀k term means s.m<maxsize in the second term of the hypothesis. The second term of I_c holds from I_c and the ∀k term. The first premise of the <u>first</u> rule becomes

$$\beta_{in} \land I_c \land 1\leq p\leq s.m \land (\forall k \in [1..p-1](s.v[k]\neq x)) \land s.v[p]=x \; \{\ \} \; \beta_{out} \land I_c$$

The second term of β_{out} follows by choosing j=p. The other terms are clear.

4a. β_{in} holds
Show: $I_c \land$ cardinality({x}∪rep(v,m)) ≤ maxsize ⊃
$(\exists i \in [1..s.m]$ *st* $x=s.v[i] \lor s.m<maxsize)$
Proof: From I_c the v[i]'s are distinct. Hence cardinality(rep(v,s.m)) is s.m. If the ∃i term is false, then x ∉ rep(v,s.m) and cardinality({x}∪rep(v,m)) = 1+s.m≤maxsize, i.e., s.m<maxsize.

4b. β_{post} holds
Show: $I_c \land$ cardinality({x}∪rep(v',s.m'))≤maxsize $\land \beta_{out}$ ⊃ s = s' ∪ {x}
Proof: s = rep(s.v,s.m) = {s.v[i] | i ∈ [1..s.m]} =
{s.v'[i] | i ∈ [1..s.m']} ∪ {s.v[s.m]} = s' ∪ {x}

For the function remove

3. Concrete operation
Show: $\beta_{in} \land I_c$ { <u>first</u> p: upto(1,s.m) <u>suchthat</u> s.v[p]=x
<u>then</u> (s.v[p]←s.v[s.m]; s.m←s.m-1 } $\beta_{out} \land I_c$
Proof: The second premise of the upto <u>first</u> rule becomes

true $\land I_c \land \forall k \in [1..s.m](s.v[k]\neq x)$ { }
$(\forall j \in [1..s.m](s.v[j]\neq x)) \land (\forall i \in [1..s.m]\ \exists j \in [1..s.m](s.v'[i]\neq x \supset$
$s.v[j]=s.v'[i])) \land I_c$

The first term follows by the ∀k term. For the second term choose j=i. I_c is clear. The first premise of the <u>first</u> rule becomes

true $\land I_c \land 1\leq p\leq s.m \land (\forall k \in [1..p-1](s.v[k]\neq x)) \land s.v[p] = x$
{ s.v[p]←s.v[s.m]; s.m←s.m-1 } $\beta_{out} \land I_c$

s.m remains non-negative since s.m'≥1. The reasons for the other terms depend on p=s.m or p≠s.m. Let p=s.m. For the second term of I_c, note that {s.v[1..s.m]}-{x} = {s.v'[1..s.m'-1]} so s.v'[1..m'-1] is duplicate-free by I_c. The first term of β_{out} follows from the ∀k term. For the second term of β_{out} choose j=i. Now let p≠s.m. By I_c, {v[1..p-1,p+1..s.m'-1]} ∪ {s.v[s.m']} = {v[1..m]} is duplicate-free. The first term of β_{out} follows from I_c and s.v'[p] = x ≠ s.v'[s.m'] = s.v[p]. For the second term of β_{out} choose j=i except when i=m' in which case choose j=p.

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge \beta_{out} \supset$ s = s' - {x}

Proof: s = {s.v[i] | i ∈ [1..s.m]}. By the first term of β_{out}, x ¬∈ s and by the second term of β_{out}, y≠x ⊃ y∈s iff y∈s'. Hence s = s' - {x}.

For the function has

3. Concrete operation

Show: $\beta_{in} \wedge I_c$ { <u>first</u> p: upto(1,s.m) <u>suchthat</u> s.v[p]=x <u>then</u> b←true <u>else</u> b←false } $\beta_{out} \wedge I_c$

Proof: I_c is unchanged. The second premise of the upto <u>first</u> rule has the hypothesis ∀k ∈ [1..s.m](s.v[k]≠x), i.e., the ∃ term in β_{out} is false = b. The first premise has the hypothesis v[p]=x, i.e., choose i=p so the ∃ term is true = b.

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge \beta_{out} \supset$ b = x ∈ s'

Proof: b = ∃i ∈ [1..s.m] *st* (s.v[i]=x) = x ∈ {v'[i] | i ∈ [1..s.m'] = x ∈ s'

QED

We noted earlier that our algorithm for remove is different from Hoare's. Since our β_{in} and β_{out} can be used for Hoare's remove, the proof of his remove requires changing only step 3.

Adding Functions to "Simpleset"

Suppose now that we wanted to add other set operations such as union, intersection, and an inclusion test. We could do this either by adding each new operation to <u>form</u> "simpleset", or we could write a new <u>form</u>, say "finiteset", which extends "simpleset". In the former case we would have access to the representation of simplesets, but we would have to be very concerned about possible side effects on the representation and about the possibility of compromising the existing verification. In addition, each such change alters the specifications of "simpleset", and thus potentially requires reverification of the programs that use "simplesets". The latter choice substantially reduces the reverification responsibilities and allows a number of users to write extended operation sets without interfering with each other. However, it is feasible only if the set of operations provided by "simpleset" is rich enough.

The version of "simpleset" presented in the previous section is not quite rich enough for extended operation sets to be independent. The chief deficiency is that there is no way for a user to find out what elements are in a set. We will remedy that by adding a generator "inset" to the simpleset form and then write an extension form "finiteset".

"Inset": a Set Element Generator

We said above that a generator produces a sequence of elements. Since sets are not inherently ordered, we can generate the elements in any order that is convenient. We do, however, want to be able to promise that each element in a set appears exactly once in the generated sequence. It is not necessary (or particularly desirable) that the elements of two equal sets be generated in the same order. In fact, the order in which this generator produces the set elements is an accident of the history of the set.[14]

The following program text is the definition of a generator, "inset", which produces the desired sequence; it is shown in its proper context within the "simpleset" form. We have, however, deleted (and replaced by ellipses) those parts of "simpleset" which are identical to their previous definition. The form inset satisfies the standard aggregate assumptions, so we specify it by giving its proof rules. For simplicity, we provide only the first and the pure for rules.

```
form simpleset(maxsize:integer, thing:form<←,=>) =
   beginform

   specifications
      . . .
      generator inset(s:simpleset) extends x:thing
         let inset = { x st x ∈ s } where s ≠ {} ⊃ (inset = q ∪ {x} ∪ r and
                                          q, {x}, and r are disjoint);
         rule for(I, x, s, ST(x, s, z)) =
            premise q ⊂ s ∧ x ∈ s-q ∧ I(q) { ST(x,s,z) } I(q ∪ {x});
```

[14] We could, of course, go to extra trouble to generate the elements in a standard order, but that is a different design decision and leads to a different program.

```
        rule first(P, x, s, β, S1(x, s, z), S2(s, z), Q) =
            premise q ⊂ s ∧ x ∈ s-q ∧ P ∧ (∀w ∈ q ¬β(w)) ∧ β(x) { S1(x,s,z) } Q,
            premise P ∧ ∀w ∈ s ¬β(w) { S2(s,z) } Q;
. . .
implementation
    . . .
    body inset =
        beginform
        representation
            unique j:integer;
            rep(s.v,s.m,x,j) = if s.m=0 then {} else q ∪ {x} ∪ r where
                    q = {s.v[i] | i ∈ [1..j-1]} and
                    x = s.v[j] and
                    r = {s.v[i] | i ∈ [j+1..m]};
            invariant true;
        implementation
            body &init out ((&b ≡ s.m>0) ∧ (&b ⊃ 1=&g.j≤s.m ∧ x=s.v[&g.j])) =
                if s.m > 0 then (&g.j←1; x←s.v[1]; &b←true)
                        else &b←false;

            body &next in 1≤&g.j≤s.m out ((&b ≡ &g.j' < s.m) ∧
                (&b ⊃ &g.j=&g.j'+1 ∧ 1≤&g.j≤s.m ∧ x=s.v[&g.j])) =
                if &g.j < s.m then (&g.j←&g.j+1; x←s.v[&g.j]; &b←true)
                        else &b←false;
        endform
endform
```

The generator "inset" can now be used to express the iteration which was posed as the first problem in the introduction, that is, to compute the sum of the elements in a set s. Compare this Alphard statement with the three versions in contemporary languages given there:

```
sum ← 0; for x:inset(s) do sum ← sum + x
```

This version of the loop *does not reveal* the implementation, so the users need not be concerned with which kind of iteration is most appropriate. In addition, the implementor of the "simpleset" form can now be reasonably sure that a change in the implementation will not create havoc in user programs. We can verify this program segment using the pure for rule for inset given in the specifications.

Show: true { sum←0; for x : inset(s) do sum←sum+x } sum = $\text{SIGMA}_{j \in s}(j)$

Proof: I({}) is sum = 0, I(q) is sum = $\text{SIGMA}_{j \in q}(j)$, and the premise of the for rule is

$q \subset s \wedge x \in s\text{-}q \wedge I(q)$ { sum←sum+x } $I(q \cup \{x\})$

This reduces to the provable formula

$q \subset s \wedge x \in s\text{-}q \wedge sum = \text{SIGMA}_{j \in q}(j) \supset sum + x = \text{SIGMA}_{j \in q \cup \{x\}}(j)$

QED

We next verify inset. We must first reconstruct the pre and post conditions for &init and &next from the specified proof rules:

&init

post (&b≡s≠{}) ∧ (&b ⊃ x ∈ s ∧ q={})

&next

pre x∈s

post (&b≡r'≠{}) ∧ (&b ⊃ x ∈ r' ∧ q=q'∪{x'})

The reasons that parts (c) and (d) of the basic generator assumptions hold are essentially the same as for upto. It is also necessary to discharge the standard aggregate assumptions:

(a) Sets are used.

(b) s = q ∪ {x} ∪ r when s≠{} (recall disjointness of q, {x}, and r).

(c) The pre and post conditions have the required form.

Since "m" and "v" are unchanged by inset, the I_c of simpleset still holds and will be used throughout this proof. The "s." qualifier is sometimes omitted in the interest of clarity.

For the form

1. Representation validity
   Show: true ⊃ true
   Proof: clear

2. Initialization
   Show: true { } true ∧ true
   Proof: clear

For the function &init

3. Concrete operation
   Show: true { if s.m>0 then (&g.j←1; x←s.v[1]; &b←true) else &b←false } β_{out} ∧ true
   Proof: clear by considering the two cases of the if

4a. β_{in} holds

β_{in} is true.

4b. β_{post} holds

Show: true ∧ (&b≡s.m>0) ∧ (&b ⊃ 1=&g.j≤s.m ∧ x=s.v[&g.j]) ⊃ (&b ≡ s≠{}) ∧ (&b ⊃ x ∈ s ∧ q = {})

Proof: To obtain s and q in terms of concrete variables, use the rep function. Then &b ≡ (s.m>0) ≡ {v[i] | i ∈ [1..m]}≠{} ≡ s≠{}. Suppose s.m>0, i.e., &b = true. Then &g.j=1 whence x=s.v[1] ∈ {v[i] | i ∈ [1..m]} and q = {v[i] | i ∈ [1..0]}={}.

For the function &next

3. Concrete operation

Similar to &init.3

4a. β_{in} holds

Show: x ∈ s ⊃ 1≤&g.j≤s.m

Proof: Using the rep function, x ∈ s implies v[j] ∈ {v[i] | i ∈ [1..m]}, whence 1≤&g.j≤m.

4b. β_{post} holds

Show: x ∈ s ∧ (&b≡&g.j'<s.m) ∧ (&b ⊃ &g.j=&g.j'+1∧1≤&g.j≤s.m∧x=s.v[&g.j]) ⊃ (&b ≡ {v[i] | i ∈ [j'+1..m]} ≠ {}) ∧ (&b ⊃ x ∈ {v[i] | i ∈ [j'+1..m]} ∧ {v[i] | i ∈ [1..j-1]}={v[i] | i ∈ [1..j']})

Proof: &b ≡ (&g.j'<s.m) ≡ {v[i] | i ∈ [j'+1..m]} ≠ {} by reasoning similar to 4a. The second term of the conclusion follows from 1≤&g.j=&g.j'+1≤s.m and x=s.v[&g.j].

QED

"Finiteset": an Extension of "Simpleset"

Since the simple set form defined above does not provide the usual set operations one expects (e.g., union), in this section we shall define and verify an extension of that form which provides these facilities. All of the mechanisms used in this example have been presented previously; the example does, however, provide us the opportunity to illustrate the use of the specifications of one form, "simpleset", in the verification of another. The new form definition and its proof are given below:

```
form finiteset(maxsize:integer, T:form<←,=>) extends s:simpleset(maxsize,T) =
  beginform
  specifications
     requires maxsize ≥ 0
     let finiteset = { . . . x_i . . . } where x_i is thing;
     invariant cardinality(finiteset) ≤ maxsize;
     initially finiteset = {};
     function
        union(s1,s2:finiteset(maxsize,T)) returns s3:finiteset(maxsize,T)
           pre cardinality(s1∪s2)≤maxsize
           post s3=s1 ∪ s2,
        intersect(s1,s2:finiteset(maxsize,T)) returns s3:finiteset(maxsize,T)
           post s3=s1 ∩ s2,
        includes(s1,s2:finiteset(maxsize,T)) returns b:boolean
           post b=s2 ⊆ s1;

  representation
     rep(s) = s
     invariant cardinality(s) ≤ maxsize

  implementation
        body union =
           begin
           for x:inset(s1) do insert(s3,x);
           for x:inset(s2) do insert(s3,x);
           end;
        body intersect =
           for x:inset(s1) do
              if has(s2,x) then insert(s3,x);
        body includes =
           first x:inset(s2) suchthat ¬has(s1,x) then b←false else b←true;
  endform
```

Verification of Finiteset

Since rep(s) is an identity function except for a type change from simpleset to finiteset, we shall assume $\beta_{pre} = \beta_{in}$ and $\beta_{post} = \beta_{out}$ in the proof. All the generator uses are independent of the loop bodies; specifically, s3 is changed but never generated. Note also that s3 is instantiated as a simpleset whenever it is needed for a return value, and hence is initialized to {}.

For the form

1. Representation validity
   Show: cardinality(s)≤maxsize ⊃ cardinality(s) ≤ maxsize
   Proof: clear

2. Initialization
   Show: maxsize≥0 { "s←{}" } s={} ∧ cardinality(s)≤maxsize
   Proof: The notation "s←{}" refers to the initially clause of simpleset.
   The proof is trivial.

For the function union

3. Concrete operation
   Show: cardinality(s1∪s2)≤maxsize ∧ I_c { body of union } s3=s1∪s2 ∧ I_c
   Proof: I_c remains true because it is unchanged. A loop invariant for the first for statement is s3 = q. Since cardinality(q) < cardinality(s1) ≤ cardinality(s1∪s2) ≤ maxsize, the pre condition of insert is met; the post condition says s3 = q ∪ {x} which shows s3 = q is indeed a loop invariant. Similarly, a loop invariant for the second for statement is s3 = s1 ∪ q. The first for statement is started with s3 = {}; the second for statement is started with s3 = s1 by the result of the first for statement, which is s3 = s1.

4a. β_{in} holds
   $\beta_{pre} = \beta_{in}$

4b. β_{post} holds
   $\beta_{post} = \beta_{out}$

For the function intersect

3. Concrete operation
   Show: I_c { body of intersect } s3 = s1∩s2 ∧ I_c
   Proof: A loop invariant is s3 = q ∩ s2 because if x ∈ s2 then s3∪{x} = (q∩s2)∪{x} = (q∪{x})∩s2 while if x ¬∈ s2 then s3 = q∩s2. The pre condition for insert holds because s3 = q∩s2 ⊂ s1∩s2 ⊆ s1∪s2. The initialization of s3 to {} starts the loop properly; the result is s3 = s1∩s2.

4a. and 4b. As in union.

For the function includes

3. Concrete operation

   Show: I_c { body of includes } b = s2 $\subseteq$ s1

   Proof: The second premise of the first rule has the hypothesis

   $\forall w \in s2 \neg\neg has(s1,w) = (s2 \subseteq s1) = true$. The first premise has the hypothesis $x \in s2 \wedge \neg(has(s1,x))$, i.e., $x \in s2 \wedge x \neg\in s1$ whence b = false as the body does.

4a. and 4b. As in union.

QED

A Remark on Program Size

We are aware of (and have occasionally shared) the apprehension of some of our colleagues that Alphard programs will be substantially, even unreasonably, larger than programs for similar tasks written in other languages. Early results indicate that this need not be the case. One comparison is made in [Shaw76]; we are now able to compare Hoare's "smallintset" with "simpleset".

First, let us compare this program text with Hoare's. The Alphard program, "simpleset", initially looks longer -- 32 lines to 28 for Hoare's "smallintset". "Simpleset", however, includes about 14 lines of verification assertions. With the exception of the in/out assertions, this information appears in Hoare's paper, but not in the "smallintset" program itself.

We will compare program sizes (exclusive of assertions) on the basis of the number of lexemes used, since the division into lines is arbitrary. We divided the lexemes into three categories: declarations and procedure headers, text grouping symbols like begin and end, and executable statements. We treated a qualified name as a single lexeme. We found the following:

	executable	grouping	declaration	total
"simpleset"	95	2	81	178
"smallintset"	121	12	58	191

Alphard's shorter executable text is largely attributable to the conciseness of the first statement; its larger declaration text seems to arise from the separation of specifications from procedure bodies and from the additional generality. The differences are not large enough to draw major conclusions from the data, and raw text length is hardly the major criterion for comparing languages. Nonetheless, the closeness of the numbers should serve to allay any fears that Alphard programs will necessarily be very large.

Conclusions

The ultimate goal of the Alphard project is to increase the quality and reduce the total, lifetime, cost of *real* programs. Of the many alternative approaches to this goal we have chosen one in which recent results from programming methodology and program verification are merged in a programming language design.

The key component of this merger is the introduction of a language mechanism, the form, to provide explicit support for the development of conceptual ***abstractions***. The close association between forms and our intuitive notion of abstraction seems sound on methodological grounds, for it permits the programmer to concentrate on abstractions instead of their implementations. It also seems sound in terms of current (and projected) verification technology in that it permits isolated proofs of manageable size which collectively verify the entire program.

The success of this approach to improving quality and reducing costs depends, in large measure, on the degree to which the proposed language mechanism is able to express natural abstractions. In a previous report [Wulf76a,b] we dealt with abstractions whose behavior is naturally expressed as a collection of operations defined over an abstract data structure. This is *not*, however, the full range of behaviors implicit in our understanding of the concept of "abstraction". Thus, in this report we concerned ourselves with that class of behaviors corresponding to the notion of enumerating the elements of an abstract aggregate (i.e., data structure).

The specific content of this report has dealt with two related issues: the language features for defining and using such abstractions and the development of specification and verification techniques to accompany the language features. It is reassuring to us that the existing form mechanism is adequate to capture the new class of abstractions introduced here. We also find it interesting that the forms which define generators can be specified quite naturally in terms of proof rules instead of the usual functional specifications. Despite the complexity of the full generator mechanism and associated proof rules, a chain of simplifying assumptions yields the simple rules for common types of loops in other languages; furthermore, these common loops terminate.

A number of open problems remain. The loop specialization facility in Alphard has made it possible to encapsulate iteration patterns along with other properties of an abstraction, but it has also made it awkward to write certain kinds of loops, including those which operate on only part of a structure and those in which a structure is modified by the loop which operates on it.

We may wish to eliminate many such irregular loops on methodological grounds, but others seem to be reasonable, understandable, and hence safe. For example, it seems acceptable to write loops for

- recurrence relations in which the first *k* elements of a vector are treated individually and the rest uniformly,

- operations on matrices in which the boundary values receive special treatment,

- tree walks in which data values at the nodes, but not the tree structure, are changed,

- list processing operations when the loop body is making insertions and deletions to the list from which elements are being generated, and

- operations in which the loop body may wish to request early loop termination (without the distributed cost and complexity of including the test in the while clause).

Since a generator is in fact a form, the ability to write some of these loops may be provided by defining functions other than &init and &next in the generator. Operations on the structure would then still be performed only by the generator, which could presumably keep matters in hand. The restrictions under which this is reasonable are a subject for further research. This is not, however, an acceptable general solution, for it would require the generator to provide its own versions of all interesting operations on the structures for which it generates elements.

A general solution for the problem of permitting interactions between the generator and the loop body can be found by returning to the original proof rule, without even the basic generator assumptions. This rule assumes only that &init and &next are functions provided by the generator. This solution is too general -- it is too unwieldy for any but the most intricate of interactions. We believe that a promising path for further research is the search for sets of reasonable assumptions which permit interesting interactions and also, like the two sets of assumptions made in this report, lead to vastly simplified proof rules.

Acknowledgements

We owe a great deal to our colleagues at CMU and ISI, especially Mario Barbacci, Neil Goldman, 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. We also appreciate very much the perceptive responses that a number of our colleagues have made on an earlier draft of this paper. Finally, we are grateful to Raymond Bates, David Lamb, Brian Reid, and Martin Yonke for their expert assistance with the document formatting programs.

References

[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).

[Hoare72a] C. A. R. Hoare, "A Note on the For Statement", *BIT*, 12, 1972 (pp. 334-341).

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

[Hoare73] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal", *Acta Informatica*, 2, 4, 1973 (pp. 335-355).

[Igarashi75] Shigeru Igarashi, Ralph L. London, and David C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation", *Acta Informatica*, 4, 2, 1975 (pp. 145-182).

[Jensen74] Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*, Springer-Verlag Lecture Notes in Computer Science, No. 18, 1974.

[Katz75] Shmuel Katz and Zohar Manna, "A Closer Look at Termination", *Acta Informatica*, 5, 4, 1975 (pp. 333-352).

[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.

[Luckham75] David C. Luckham and Norihisa Suzuki, "Automatic Program Verification IV: Proof of Termination within a Weak Logic of Programs", *Memo AIM-269*, Stanford University, October 1975.

[McCarthy62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, *LISP 1.5 Programmer's Manual*, MIT Press, 1962.

[Newell64] Allen Newell, Fred Tonge, Edward A. Feigenbaum, Bert F. Green Jr., and George H. Mealy, *Information Processing Language-V Manual*, Second Edition, Prentice-Hall, 1964.

[Shaw76] Mary Shaw, "Abstraction and Verification in Alphard: Design and Verification of a Tree Handler", *Proc. Fifth Texas Conference on Computing Systems*, 1976 (to appear).

[Teitelman75] Warren Teitelman, "Interlisp Reference Manual", Xerox PARC, 1975.

[Weissman67] Clark Weissman, *LISP 1.5 Primer*, Dickenson, 1967.

[Wulf76a] Wm. A. Wulf, Ralph L. London, and Mary Shaw, "Abstraction and Verification in Alphard: Introduction to Language and Methodology", *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Wulf76b] Wm. A. Wulf, Ralph L. London, and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Transactions on Software Engineering*, SE-2,4, December, 1976 (to appear).

Appendix A
Informal Description of Verification Methodology

Alphard's verification methodology is designed to determine whether a <u>form</u> will actually behave as promised by its abstract specifications. The methodology depends on explicitly separating the description of how an object behaves from the code that manipulates the representation in order to achieve that behavior. It is derived from Hoare's technique for showing correctness of data representations[Hoare72b].

The abstract object and its behavior are described in terms of some mathematical entities natural to the problem domain. Graphs are used in [Shaw76] to describe binary trees; sequences are used in [Wulf76a,b] to describe queues and stacks and in [London76] to describe list processing, and so on. We appeal to these abstract types:

- in the <u>invariant</u>, which explains that an instantiation of the <u>form</u> may be viewed as an object of the abstract type that meets certain restrictions,

- in the <u>initially</u> clause, where a particular abstract object is displayed, and

- in the <u>pre</u> and <u>post</u> conditions for each function, which describe the effect the function has on an abstract object which satisfies the invariant.

The <u>form</u> contains a parallel set of descriptions of the concrete object and how it behaves. In many cases this makes the effect of a function much easier to specify and verify than would the abstract description alone.

Now, although it is useful to distinguish between the behavior we want and the data structures we operate on, we also need to show a relationship that holds between the two. This is achieved with the representation function <u>rep</u>(x), which gives a mapping from the concrete representation to the abstract description. The purpose of a form verification is to ensure that the two invariants and the rep(x) relation between them are preserved.

In order to verify a form we must therefore prove four things. Two relate to the representation itself and two must be shown for each function. Informally, the four required steps are[15]:

For the form

1. Representation validity

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

2. Initialization

   requires { *init clause* } initially(rep(x)) $\wedge$ $I_c(x)$

For each function

3. Concrete operation

   in(x) $\wedge$ $I_c(x)$ { *function body* } out(x) $\wedge$ $I_c(x)$

4. Relation between abstract and concrete

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

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

Step 1 shows that any legal state of the concrete representation has a corresponding abstract object (the converse is deducible from the other steps). Step 2 shows that the initial state created by the representation section is legal. Step 3 is the standard verification formula for the concrete operation as a simple program; note that it enforces the preservation of I_c. Step 4 guarantees (a) that the concrete operation is applicable whenever the abstract pre condition holds and (b) that if the operation is performed, the result corresponds properly to the abstract specifications.

Appendix B
Derivations of Simplified Proof Rules

In this Appendix we show that the general for and first proof rules and the basic

[15] We will use $I_a(rep(x))$ to denote the abstract invariant of an object whose concrete representation is x, $I_c(x)$ to denote the corresponding concrete invariant, italics to refer to code segments, and the names of specification clauses and assertions to refer to those formulas. In step 4b, "pre(rep(x'))" refers to the value of x *before* execution of the function. A complete development of the form verification methodology appears in [Wulf76a,b].

generator assumptions yield the simplified proof rules based on those assumptions. We shall use the following two sets of assumptions and three proof rules:

Generator Assumptions

(G1) $G \wedge \beta_{init.pre}$ { $\pi \leftarrow$ x.&init } $G \wedge \beta_{init.post}$
(G2) $G \wedge \beta_{next.pre}$ { $\pi \leftarrow$ x.&next } $G \wedge \beta_{next.post}$
(G3) β_{req} { *init clause* } G

Basic Generator Assumptions

(BG1) The post conditions on &init and &next are of the form

$$(b \equiv \pi_i) \wedge \beta_i \quad \text{and} \quad (b \equiv \pi_n) \wedge \beta_n$$

respectively, where b is the result parameter of these functions.

(BG2) $G \supset \beta_{init.pre}$, $G \wedge (\pi_i \wedge \beta_{init.post} \vee \pi_n \wedge \beta_{next.post}) \supset \beta_{next.pre}$

(BG3) The generator and the loop body are independent. That is, for arbitrary predicates R and S

R(y,z) { *init clause* } R(y,z)
R(y,z) { $\pi \leftarrow$ x.&init } R(y,z)
R(y,z) { $\pi \leftarrow$ x.&next } R(y,z)
and S(x,y) { ST(x,y,z) } S(x,y)

And Rule

$$\frac{P_1 \{ S \} Q_1,\ P_2 \{ S \} Q_2}{P_1 \wedge P_2 \{ S \} Q_1 \wedge Q_2}$$

Consequence Rules

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

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

Semicolon Rule

$$\frac{P \{ S_1 \} Q, Q \{ S_2 \} R}{P \{ S_1; S_2 \} R}$$

Let us work initially on the for statement. Its general proof rule is

(Gfor0) $P \wedge \beta_{req}$ { *init clause* } $P \wedge \beta_{init.pre}$

(Gfor1) $P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow$ x.&init } $\neg(\pi \wedge \beta(x)) \supset Q$

(Gfor2) $P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow$ x.&init; assume $\pi \wedge \beta(x)$; ST(x,y,z) } $I \wedge G \wedge \beta_{next.pre}$

(Gfor3) $I \wedge G \wedge \beta_{next.pre}$ { $\pi \leftarrow$ x.&next } $\neg(\pi \wedge \beta(x)) \supset Q$

(Gfor4) $I \wedge G \wedge \beta_{next.pre}$ { $\pi \leftarrow$ x.&next; assume $\pi \wedge \beta(x)$; ST(x,y,z) } $I \wedge G \wedge \beta_{next.pre}$

$P \wedge \beta_{req}$ { for x: gen(y) while $\beta(x)$ do ST(x,y,z) | I } Q

and the simplified proof rule is

(Sfor1, Sfor2) $G \wedge [P \wedge \beta_i \wedge \neg(\pi_i \wedge \beta(x)) \vee I \wedge \beta_n \wedge \neg(\pi_n \wedge \beta(x))] \supset Q$

(Sfor3, Sfor4) $G \wedge \beta(x) \wedge [P \wedge \beta_i \wedge \pi_i \vee I \wedge \beta_n \wedge \pi_n]$ { ST(x,y,z) } I

$P \wedge \beta_{req}$ { for x: gen(y) while $\beta(x)$ do ST(x,y,z) | I } Q

Our task, therefore, is to derive each of the five Gfor premises from G, BG, and the four Sfor premises. If we do this, we obtain the conclusion of the general rule which is the conclusion of the simplified rule. Note that the *init clause* in Gfor0 is invoked when the generator is instantiated by the clause "local x:gen(y)" in the expansion of the for statement.

We first note relationships involving x.&next, x.&init, the invariant I, and the assertion P. Assumption BG1 means that for an arbitrary predicate R involving the set of generated values $x_0,\ldots,x_p$, and x (in this notation x is also denoted by x_{p+1}), we know

$$R(\{x_0,\ldots,x_p,x\}) \{ \pi \leftarrow x.\&next \} R(\{x_0,\ldots,x_p,x_{p+1}\}) \wedge (\pi_n \supset x=x_{p+2})$$
$$R(\{\}) \{ \pi \leftarrow x.\&init \} R(\{\}) \wedge (\pi_i \supset x=x_0)$$

Thus, provided x is denoted by x_{p+1}, the predicate R is preserved by x.&next and x.&init, and there may be a newly generated value. Using both BG1 and BG3 we see that x.&next preserves the invariant I, which depends on x, y, and z. The cases of the *init clause* and x.&init preserving P are simpler since P depends only on y and z.

Derivation of Gfor0

β_{req} { *init clause* } G	G3
P { *init clause* } P	BG3
$P \wedge \beta_{req}$ { *init clause* } $P \wedge G$	and rule
$G \supset \beta_{init.pre}$	BG2
$P \wedge \beta_{req}$ { *init clause* } $P \wedge \beta_{init.pre}$	consequence

Derivation of Gfor1

$G \wedge \beta_{init.pre}$ { $\pi \leftarrow x.\&init$ } $G \wedge \beta_{init.post}$	G1
P { $\pi \leftarrow x.\&init$ } P	BG3
$P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow x.\&init$ } $G \wedge P \wedge \beta_{init.post}$	and rule
$G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i \supset \neg(\pi \wedge \beta(x)) \supset Q$	Sfor1
$P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow x.\&init$ } $\neg(\pi \wedge \beta(x)) \supset Q$	consequence, BG1

Derivation of Gfor2

$P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow x.\&init$ } $G \wedge P \wedge \beta_{init.post}$	step 3 above
$G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i \wedge \pi_i \wedge \beta(x) \supset G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i \wedge \pi_i \wedge \beta(x)$	identity
$G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i$ { $\underline{assume}\ \pi \wedge \beta(x)$ } $G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i \wedge \pi_i \wedge \beta(x)$	$\underline{assume}$ rule
$G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i \wedge \pi_i \wedge \beta(x)$ { $ST(x,y,z)$ } $I \wedge \pi_i$	Sfor3, private π_i
$G \wedge \beta_{init.post}$ { $ST(x,y,z)$ } $G \wedge \beta_{init.post}$	BG3
$G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i \wedge \pi_i \wedge \beta(x)$ { $ST(x,y,z)$ } $I \wedge G \wedge \pi_i \wedge \beta_{init.post}$	and rule
$G \wedge P \wedge (\pi \equiv \pi_i) \wedge \beta_i$ { $\underline{assume}\ \pi \wedge \beta(x);\ ST(x,y,z)$ } $I \wedge G \wedge \pi_i \wedge \beta_{init.post}$	semicolon rule
$G \wedge \pi_i \wedge \beta_{init.post} \supset \beta_{next.pre}$	BG2
$P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow x.\&init;\ \underline{assume}\ \pi \wedge \beta(x);\ ST(x,y,z)$ } $I \wedge G \wedge \beta_{next.pre}$	semicolon rule, consequence, BG1

Derivation of Gfor3

$G \wedge \beta_{next.pre}$ { $\pi \leftarrow x.\&next$ } $G \wedge \beta_{next.post}$	G2
I { $\pi \leftarrow x.\&next$ } I	BG1, BG3
$I \wedge G \wedge \beta_{next.pre}$ { $\pi \leftarrow x.\&next$ } $G \wedge I \wedge \beta_{next.post}$	and rule
$G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n \supset \neg(\pi \wedge \beta(x)) \supset Q$	Sfor2
$I \wedge G \wedge \beta_{next.pre}$ { $\pi \leftarrow x.\&next$ } $\neg(\pi \wedge \beta(x)) \supset Q$	consequence, BG1

Derivation of Gfor4

$I \wedge G \wedge \beta_{next.pre}$ { $\pi \leftarrow x.\&next$ } $G \wedge I \wedge \beta_{next.post}$	step 3 above
$G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n \wedge \pi_n \wedge \beta(x) \supset G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n \wedge \pi_n \wedge \beta(x)$	identity
$G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n$ { $\underline{assume}\ \pi \wedge \beta(x)$ } $G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n \wedge \pi_n \wedge \beta(x)$	$\underline{assume}$ rule
$G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n \wedge \pi_n \wedge \beta(x)$ { $ST(x,y,z)$ } $I \wedge \pi_n$	Sfor4, private π_n
$G \wedge \beta_{next.post}$ { $ST(x,y,z)$ } $G \wedge \beta_{next.post}$	BG3
$G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n \wedge \pi_n \wedge \beta(x)$ { $ST(x,y,z)$ } $I \wedge G \wedge \pi_n \wedge \beta_{next.post}$	and rule
$G \wedge I \wedge (\pi \equiv \pi_n) \wedge \beta_n$ { $\underline{assume}\ \pi \wedge \beta(x);\ ST(x,y,z)$ } $I \wedge G \wedge \pi_n \wedge \beta_{next.post}$	semicolon rule
$G \wedge \pi_n \wedge \beta_{next.post} \supset \beta_{next.pre}$	BG2
$I \wedge G \wedge \beta_{next.pre}$ { $\pi \leftarrow x.\&next;\ \underline{assume}\ \pi \wedge \beta(x);\ ST(x,y,z)$ } $I \wedge G \wedge \beta_{next.pre}$	semicolon rule, consequence, BG1

We now work on the first statement. The expansion of

$$\underline{\text{first}}\ x{:}gen(y)\ \underline{\text{suchthat}}\ \beta(x)\ \underline{\text{then}}\ S_1(x,y,z)\ \underline{\text{else}}\ S_2(y,z)\ \}\ Q$$

using a standard while statement, including the most general case assertions, is

```
assert P ∧ β_req;
begin label λ;
    begin local x: gen(y);
        assert P ∧ G ∧ β_init.pre;
        π ← x.&init;
        while
            [assert P ∧ G ∧ ¬β(x_0..x_p) ∧ (π⊃β_next.pre) ∧ (β_init.post ∨ β_next.post)]
            π do
                if β(x) then (S_1(x,y,z); goto λ) else π ← x.&next
    end;
    S_2(y,z);
λ: end;
assert Q
```

The general proof rule for the first statement is

(Gfirst0) $P \wedge \beta_{req}$ { *init clause* } $P \wedge \beta_{init.pre}$

(Gfirst1) $P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow x.\&init$ } $P \wedge G \wedge (\pi \supset \beta_{next.pre})$

(Gfirst2) $P \wedge G \wedge \neg\beta(x_0..x_p) \wedge (\beta_{init.post} \vee \beta_{next.post}) \wedge \beta_{next.pre} \wedge \pi \wedge \beta(x)$ { $S_1(x,y,z)$ } Q

(Gfirst3) $P \wedge G \wedge \neg\beta(x_0..x_p) \wedge (\beta_{init.post} \vee \beta_{next.post}) \wedge \neg\pi$ {$S_2(y,z)$ } Q

(Gfirst4) $P \wedge G \wedge \neg\beta(x_0..x_p) \wedge \beta_{next.pre} \wedge \neg\beta(x)$ { $\pi \leftarrow x.\&next$ } $P \wedge G \wedge \neg\beta(x_0..x_{p+1}) \wedge (\pi \supset \beta_{next.pre})$

$P \wedge \beta_{req}$ { first x:gen(y) suchthat $\beta(x)$ then $S_1(x,y,z)$ else $S_2(y,z)$ } Q

and the simplified proof rule is

(Sfirst1) $G \wedge P \wedge [\beta_i \wedge \pi_i \vee \beta_n \wedge \pi_n \wedge \neg\beta(x_0..x_p)] \wedge \beta(x)$ { $S_1(x,y,z)$ } Q

(Sfirst2) $G \wedge P \wedge [\neg\pi_i \wedge \beta_i \vee \neg\pi_n \wedge \beta_n \wedge \neg\beta(x_0..x_p)]$ { $S_2(y,z)$ } Q

$P \wedge \beta_{req}$ { first x: gen(y) suchthat $\beta(x)$ then $S_1(x,y,z)$ else $S_2(y,z)$ } Q

In Gfirst1 note that there is no x_p before the statement $\pi \leftarrow x.\&init$ so $\neg\beta(x_0..x_p) \equiv true$. As in the for case, the task is to derive each of the five Gfirst premises from G, BG, and the two Sfirst premises.

Derivation of Gfirst0

Same as derivation of Gfor0

Derivation of Gfirst1

$G \wedge \beta_{init.pre}$ { $\pi \leftarrow$ x.&init } $G \wedge \beta_{init.post}$	G1
P { $\pi \leftarrow$ x.&init } P	BG3
$P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow$ x.&init } $G \wedge P \wedge \beta_{init.post}$	and rule
$G \wedge P \wedge \beta_{init.post} \supset P \wedge G \wedge (\pi \supset \beta_{next.pre})$	BG2
$P \wedge G \wedge \beta_{init.pre}$ { $\pi \leftarrow$ x.&init } $P \wedge G \wedge (\pi \supset \beta_{next.pre})$	consequence

Derivation of Gfirst2

$G \wedge P \wedge [\beta_i \wedge \pi_i \wedge true \vee \beta_n \wedge \pi_n \wedge \neg\beta(x_0..x_p)] \wedge \beta(x)$ { $S_1(x,y,z)$ } Q	Sfirst1
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge [(\pi \equiv \pi_i) \wedge \beta_i \vee (\pi \equiv \pi_n) \wedge \beta_n] \wedge \pi \wedge \beta(x)$ { $S_1(x,y,z)$ } Q	algebra
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge (\beta_{init.post} \vee \beta_{next.post}) \wedge \pi \wedge \beta(x)$ { $S_1(x,y,z)$ } Q	BG1
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge (\beta_{init.post} \vee \beta_{next.post}) \wedge \beta_{next.pre} \wedge \pi \wedge \beta(x)$ { $S_1(x,y,z)$ } Q	consequence

Derivation of Gfirst3

$G \wedge P \wedge [\neg\pi_i \wedge \beta_i \wedge true \vee \neg\pi_n \wedge \beta_n \wedge \neg\beta(x_0..x_p)]$ { $S_2(y,z)$ } Q	Sfirst2
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge [(\pi \equiv \pi_i) \wedge \beta_i \vee (\pi \equiv \pi_n) \wedge \beta_n] \wedge \neg\pi$ { $S_2(y,z)$ } Q	algebra
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge (\beta_{init.post} \vee \beta_{next.post}) \wedge \neg\pi$ { $S_2(y,z)$ } Q	BG1

Derivation of Gfirst4

$G \wedge \beta_{next.pre}$ { $\pi \leftarrow$ x.&next } $G \wedge \beta_{next.post}$	G2
P { $\pi \leftarrow$ x.&next } P	BG3
$\neg\beta(x_0..x_p) \wedge \neg\beta(x)$ { $\pi \leftarrow$ x.&next } $\neg\beta(x_0..x_{p+1})$	BG1, definition of $\neg\beta(x_0..x_p)$
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge \neg\beta(x) \wedge \beta_{next.pre}$ { $\pi \leftarrow$ x.&next } $P \wedge G \wedge \beta_{next.post} \wedge \neg\beta(x_0..x_{p+1})$	and rule
$P \wedge G \wedge \neg\beta(x_0..x_p) \wedge \beta_{next.pre} \wedge \neg\beta(x)$ { $\pi \leftarrow$ x.&next } $P \wedge G \wedge \neg\beta(x_0..x_{p+1}) \wedge (\pi \supset \beta_{next.pre})$	BG2, consequence