memory, we would like to summarize our experiences with the following observations and cautions.

(1) The benefits from these linguistic mechanisms, large though they might be, do not come automatically. A programmer must learn to use them effectively. We are just beginning to learn how to do so.

(2) Just as the absence of *goto*s does not always make a program better, the absence of type errors does not make it better if their absence is purchased by sacrificing clarity, efficiency, or type articulation.

(3) Most good programmers use many of the techniques implied by these disciplines, often subconsciously, and can do so in any reasonable language. Language design can help by making the discipline more convenient and systematic, and by catching blunders or other unintended violations of conventions. Acquiring a particular programming style seems to depend on having a language that supports or requires it; once assimilated, however, that style can be applied in many other languages.

Acknowledgments. The principal designers of Mesa, in addition to the authors, have been Butler Lampson and Jim Mitchell. The major portion of the Mesa operating system was programmed by Richard Johnsson and John Wick of the System Development Division of Xerox. In addition to those mentioned above, Douglas Clark, Howard Sturgis, and Niklaus Wirth have made helpful comments on earlier versions of this paper.

References

1. Dahl, O.-J., Myhrhaug, B., and Nygaard, K. The SIMULA 67 common base language. Publ. No. S-2, Norwegian Comptng. Ctr., Oslo, May 1968.
2. Dennis, J.B., and Van Horn, E. Programming semantics for multiprogrammed computations. *Comm. ACM 9,* 3 (March 1966), 143-155.
3. Geschke, C., and Mitchell, J. On the problem of uniform references to data structures. *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 207-219.
4. Habermann, A.N. Critical comments on the programming language PASCAL. *Acta Informatica 3* (1973), 47-57.
5. Knuth, D. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
6. Koster, C.H.A. On infinite modes. *ALGOL Bull. AB 30.3.3* (Feb. 1969), 109-112.
7. Lampson, B., Mitchell, J., and Satterthwaite, E. On the transfer of control between contexts. In *Lecture Notes in Computer Science, Vol. 19*, G. Goos and J. Hartmanis, Eds., Springer-Verlag, New York. (1974), 181-203.
8. Mitchell, J., and Wegbreit, B. Schemes: a high level data structuring concept. To appear in *Current Trends in Programming Methodologies,* R. Yeh, Ed., Prentice-Hall, Englewood Cliffs, N.J.
9. Morris, J. Protection in programming languages. *Comm. ACM 16*, 1 (Jan 1973), 15-21.
10. Parnas, D. A technique for software module specification. *Comm. ACM 15*, 5 (May 1972), 330-336.
11. Stoy, J.E., and Strachey, C. OS6 – an experimental operating system for a small computer, Part 2; input/output and filing system. *Computer J. 15*, 3 (Aug 1972), 195-203.
12. van Wijngaarden, A., Ed. A report on the algorithmic language ALGOL 68. *Num. Math. 14*, 2 (1969), 79-218.
13. Wegbreit, B. The treatment of data types in EL1. *Comm. ACM 17*, 5 (May 1974), 251-264.
14. Wirth, N. The programming language PASCAL. *Acta Informatica 1* (1971), 35-63.

Language Design for Reliable Software

S.L. Graham Editor

Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators

Mary Shaw and William A. Wulf
Carnegie-Mellon University

Ralph L. London
University of Southern California

The Alphard "form" provides the programmer with a great deal of control over the implementation of abstract data types. In this paper the abstraction techniques are extended 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. A means of specializing Alphard's loops to operate on abstract entities without explicit dependence on the representation of those entities is introduced. Specification and verification techniques that allow the properties of the generators for such iterations to be expressed in the form of proof rules are developed. Results are obtained that for common special cases of these loops are essentially identical to the corresponding constructs in other languages. A means of showing that a generator will terminate is also provided.

Key Words 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

CR Categories: 4.20, 5.24

Research supported in part by NSF grant DCR 74-04187 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.

A version of this paper was presented at the SIGPLAN/SIGOPS/SIGSOFT Conference on Language Design for Reliable Software, Raleigh, N.C. March 28-30, 1977.

Authors' addresses: M. Shaw and W.A. Wulf, Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213; R.L. London, University of Southern California Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291.

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 [16, 17], particularly the use of **forms** for abstraction and the verification methodology for **forms.** The summary of the verification methodology in the Appendix also provides the reader with a brief glimpse of **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 knew that the set elements all lie in the range [lb..ub], 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 [11] and in Lisp as the mapping functions [10, 15].

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 [16, 17].

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 paper 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 construction 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 [14] contains a wide variety of iteration statements, one of which specializes to this construct.)

We obtain general proof rules for the two loop constructs and 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 [4, 6]) and to simple rules for a substantial number of interesting abstract structures (e.g. those given by Hoare [2]).

Finally, we 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.

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 [1].) 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 is an illustration of 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 ned 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 [16, 17] 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 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]

[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(y)" 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; for simplicity we write β(x) instead of the more exact β(x, y, z).

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 ∈ 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

```
begin local x: gen(y), π: boolean;
  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 programmer.

One of the generators defined in Alphard's 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** β(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

```
begin label λ;
  begin local x: gen(y), π: boolean;
    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 [16] 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 by 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 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 show how a generator is defined and invoked, still using

[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** S_1" or "**else** S_2" may be omitted; an omitted clause is assumed to denote the empty statement.

[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 [16, 17].

"upto" as an example. We first present its definition, then add assertions, verify it as a **form**, and establish its special properties as a generator.

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 + κ; κ ← x + 1);
end
```

Since "upto" is a **form**, we can verify the **form** properties as described in [16, 17] and summarized in the Appendix. With verification information added in italics, the definition of "upto" becomes

```
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 shall 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. The notation k' denotes the value of k upon entry to &next.

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)

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

Proof: With the assignment axiom, the expression becomes
$lb \le k \le ub \supset (k + 1 \le ub \equiv k' < ub) \wedge$
$(k + 1 \le ub \supset k + 1 = k' + 1 \wedge lb \le k + 1 \le ub)$
which holds because $k' = k$ is an implicit hypothesis of the antecedent.

4. Relation between abstract and concrete
Same as &init.4.

Proof Rules for Loops

In this section we consider the verification of Alphard's two iteration constructs, **for** and **first**. Specifically, we develop proof rules for these statements, discovering in the process certain desirable properties for **forms** 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 consider a proof rule for the **for** 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 make a small number of basic assumptions about the generator. For the 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 **form**. They will allow us to simplify substantially the proof rules for the **for** and **first** statements. Next we 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 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

$P\{$**for** x: gen(y) **while** $\beta(x)$ **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 also assume that $\beta(x)$ has no side effects. We 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.[7]

$\beta_{f,j}$ = 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.

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

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

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

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

The expansion of

for x: gen(y) **while** $\beta(x)$ **do** ST(x,y,z)

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), π: boolean;
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 shall 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* $\beta(x)$ appears between either the statements $\pi \leftarrow$ x.&init or $\pi \leftarrow$ x.&next and ST(x,y,z). To indicate in these paths that π *cand* $\beta(x)$ may be assumed between the statements, the **assume** clause is introduced.[8] Its proof rule is

$$\frac{P \wedge Q \supset R}{P\ \{\textbf{assume}\ Q\}\ R}.$$

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

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

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

[8] The **assume** clause appears in [5, 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 *init clause*, the two 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 *init clause* and 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 a good deal.

Basic Generator Assumptions

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

$(b \equiv \pi_i) \wedge \beta_i$ 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 [16, 17], 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 **for** statement[9]:

$$\frac{G \wedge [P \wedge \beta_i \wedge \neg(\pi_i \wedge \beta(x)) \vee I \wedge \beta_n \wedge \neg(\pi_n \wedge \beta(x))] \supset Q \qquad G \wedge \beta(x) \wedge [P \wedge \beta_i \wedge \pi_i \vee I \wedge \beta_n \wedge \pi_n]\,\{ST(x,y,z)\}I}{P \wedge \beta_{req}\,\{\textbf{for } x\text{: gen(y) } \textbf{while } \beta(x) \textbf{ do } ST(x,y,z) \mid I\}Q}$$

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 **first** statement:

$$\frac{G \wedge P\,[\beta_i \wedge \pi_i \vee \beta_n \wedge \pi_n \wedge \neg\beta(x_0..x_p)] \wedge \beta(x)\,\{S_1(x,y,z)\}Q \qquad G \wedge P \wedge [\neg\pi_i \wedge \beta_i \vee \neg\pi_n \wedge \beta_n \wedge \neg\beta(x_0..x_p)]\,\{S_2(y,z)\}Q}{P \wedge \beta_{req}\{\textbf{first } x\text{: gen(y) } \textbf{suchthat } \beta(x) \textbf{ then } S_1(x,y,z) \textbf{ else } S_2(y,z)\}Q}$$

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 **suchthat** 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 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,
$\langle\,\rangle$ = 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 { }, $\langle\,\rangle$, and [ ]; the corresponding @ operators are union, concatenation, and merging adjacent intervals.

[9] The justifications of this and the **first** rule, from the corresponding general rules and the basic generator assumptions, are given in [13].

(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 $\langle x \rangle$ 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, $\langle x \rangle$, and t are mutually disjoint.

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

functions
  &init(&g: gen) **returns** &b:boolean
    post $(\&b \equiv T \neq \langle\rangle) \wedge (\&b \supset x = \text{lead}(T) \wedge D_1(x))$
  &next(&g: gen) **returns** &b:boolean
    pre $D_2(x)$
    post $(\&b \equiv t' \neq \langle\rangle) \wedge (\&b \supset x = \text{lead}(t') \wedge D_3(x))$

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 **for** statement becomes

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

and the **first** rule simplifies to

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

We call these two rules the standard aggregate rules.

Special Cases and Examples

The Pure "for" Rule

In many cases the programmer may wish to drop the **while** 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 **for** statements given by Hoare [2]:

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

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 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 part, note that no means is provided for the user of the **form** 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 $lb \leq k \leq ub$.

(c) **pre** and **post** conditions have the required form. Substituting the interval definitions into the standard aggregate rules and simplifying, we obtain

$$\frac{\begin{array}{l} P \wedge (lb > ub \vee \neg\beta(lb)) \vee lb \leq k \leq ub \wedge I[lb..k-1] \\ \qquad \wedge \neg\beta(k) \vee lb \leq ub \wedge I[lb..ub] \supset Q \\ lb \leq ub \wedge (P \wedge \beta(lb) \vee lb \leq k \leq ub \wedge I[lb..k-1] \\ \qquad \wedge \beta(k))\ \{ST(k,y,z)\}\ I[lb..k] \end{array}}{P\ \{\textbf{for}\ k\text{: upto(lb, ub) } \textbf{while}\ \beta(k)\ \textbf{do}\ ST(k,y,z) \mid I(k,y,z)\}\ Q}$$

and

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

where the y parameters are $\langle lb,ub \rangle$. In the special case $P = I[]$, $Q = I[lb..ub]$, and $\beta \equiv$ true, we obtain the Pascal rule for the **for** statement [2, 4]:

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

As must be the case, this rule is also obtained from the pure **for** 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 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

$$\text{true} \wedge [P \wedge (\text{false} \vee \neg\beta(\text{true})) \vee \text{true} \wedge I(\text{true}^*) \wedge (\text{false} \vee \neg\beta(\text{true}))] \supset Q$$

$$\frac{\text{true} \wedge [P \wedge \beta(\text{true}) \vee \text{true} \wedge I(\text{true}^*) \wedge \beta(\text{true})]\ \{ST(\text{true},,z)\}\ I(\text{true}^*)}{P\ \{\textbf{for}\ x\text{: forever}\ \textbf{while}\ \beta(\text{true})\ \textbf{do}\ ST(\text{true},,z) \mid I(\text{true}^*)\}\ Q}$$

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

$$\frac{I \wedge \beta\{ST(z)\}\ I}{I\{\textbf{while}\ \beta\ \textbf{do}\ ST(x)\}\ I \wedge \neg\beta}$$

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 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] ≡$_{df}$ ⟨a,a + j,a + 2 * j, ... , b − (b − a) mod j⟩ **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] provided (b − a) mod 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 illustrate the use of the proof rules given above by verifying the "eqvecs" function given earlier. With **pre** and **post** 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
```

If the upto **first** rule is used, the proof requires that we establish the two premises:

Show: true ∧ lb ≤ i ≤ ub ∧ (∀w ∈ [lb..i − 1] ¬ (A[y] ≠ B[y])) ∧ A[i] ≠ B[i] {eq → false} eq ≡ ∀j ∈ [lb..ub] A[j] = B[j].
Proof: This simplifies to lb ≤ i ≤ ub ∧ A[i] ≠ B[i] ⊃ ∃j ∈ [lb..ub] A[j] ≠ B[j]. Choose j = i.
Show: true ∧ ∀w ∈ [lb..ub] ¬ (A[w] ≠ B[w]) {eq → true} eq ≡ ∀j ∈ [lb..ub] A[j] = B[j].
Proof: clear.

Comparison with the Lisp Mapping Functions

Other examples of generators, in addition to those in this paper, may be found in [8] and [13]. In particular, the latter paper contains a generator of the elements from integer sequences where the sequences are represented by a restricted, but not uncommon, style of list processing. This generator may be easily and directly transformed into generators for use in Alphard iteration statements that express (some of) the Lisp mapping functions. The two functions map and mapc, which generate from the input list the tails or cdrs, and the elements or cars, respectively, are expressible as **for** statements whose body is the functional argument to map or mapc. The collecting, or consing, in maplist and mapcar must be done by the body of the **for** statement.

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.[10] 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

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 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 shall have shown that use of the generator cannot cause nontermination for any body.

Consider the statement

i ← 0; **for** x: gen(y) **do** i ← i + 1

If we could find: (1) a (nonnegative) value M_y depending only on y for which i ≤ 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 ≤ M_y ∧ 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[11]

i = 0 {**for** x: gen(y) **do** i ← i + 1 | i ≤ M_y ∧ I_y(x)} i ≤ M_y

Note that the clause "i ≤ 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 ≤ size(T) ∧ 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

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

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

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 paper [16, 17], 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 paper 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 paper 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 described in this paper

[10] Note that nontermination of the loop might also be caused by nontermination 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.

[11] This method for showing termination is a simple instance of the commonly used well-founded set notion [7, 9]. Here the well-founded set is the nonnegative integers bounded by M_y.

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 an *init clause* and 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 paper, lead to vastly simplified proof rules. We also believe that some of these new assumptions can be stated in terms of additional invariant properties of the generator specifications and the operations on the structure. We expect to explore these topics in future papers.

Appendix. Informal Description of Verification Methodology

Alphard's verification methodology is designed to determine whether a **form** 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 [3].

The abstract object and its behavior are described in terms of some mathematical entities natural to the problem domain. Graphs are used in [12] to describe binary trees; sequences are used in [16] to describe queues and stacks and in [8] to describe list processing, and so on. We appeal to these abstract types:

- —in the **invariant**, which explains that an instantiation of the **form** may be viewed as an object of the abstract type that meets certain restrictions,
- —in the **initially** clause, where a particular abstract object is displayed, and
- —in the **pre** and **post** conditions for each function, which describe the effect the function has on an abstract object which satisfies the invariant.

The **form** contains a parallel set of descriptions of the concrete object and how it behaves. In many cases this makes a function's effect 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 **rep**(x), which gives a mapping from the concrete representation to the abstract description. 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[12]:

For the **form**

1. Representation validity
   $I_c(x) \supset I_a(\text{rep}(x))$
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

[12] We use $I_a(\text{rep}(x))$ 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 [16, 17].

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.

Acknowledgments. We owe a great deal to our colleagues at Carnegie-Mellon University and the University of Southern California Information Sciences Institute, 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 M.I.T., 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

1. Dahl, O.-J., and Hoare, C.A.R. Hierarchical program structures. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, New York, 1972, pp. 175–220.
2. Hoare, C.A.R. A note on the for statement. *BIT 12* (1972), 334–341.
3. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 1*, 4 (1972), 271–281.
4. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language Pascal. *Acta Informatica* 2, 4 (1973), 335–355.
5. Igarashi, S., London, R.L., and Luckham, D.C. Automatic program verification I: a logical basis and its implementation. *Acta Informatica 4*, 2 (1975), 145–182.
6. Jensen, K., and Wirth, N. *PASCAL User Manual and Report.* Lecture Notes in Computer Science, No. 18, Springer-Verlag, 1974.
7. Katz, S., and Manna, Z. A closer look at termination. *Acta Informatica 5*, 4 (1975), 333–352.
8. London, R.L., Shaw, M., and Wulf, W.A. Abstraction and verification in Alphard: a symbol table example. Tech. Reports, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., and Carnegie-Mellon U., Pittsburgh, Pa., 1976.
9. Luckham, D.C., and Suzuki, N. Automatic program verification IV: Proof of termination within a weak logic of programs. Memo AIM-269, Stanford University, Stanford, Calif., Oct. 1975.
10. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.
11. Newell, A., Tonge, F., Feigenbaum, E.A., Green, B.F. Jr., and Mealy, G.H. *Information Processing Language-V Manual.* Prentice-Hall, Englewood Cliffs, N.J., Sec. Ed. 1964.
12. Shaw, M. Abstraction and verification in Alphard: design and verification of a tree handler. Proc. Fifth Texas Conf. on Computing Systems, 1976, pp. 86–94.
13. Shaw, M., Wulf, W.A., and London, R.L. Abstraction and verification in Alphard: Iteration and generators. Tech. Reports, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., and Carnegie-Mellon U., Pittsburgh, Pa., 1976.
14. Teitelman, W. Interlisp Reference Manual. Xerox Palo Alto Res. Ctr., Palo Alto, Calif., 1975.
15. Weissman, C. *LISP 1.5 Primer,* Dickenson, Encino, Calif., 1967.
16. Wulf, W.A., London, R.L., and Shaw, M. Abstraction and verification in Alphard: Introduction to language and methodology. Tech. Reports, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., and Carnegie-Mellon U., Pittsburgh, Pa., 1976.
17. Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. on Software Eng. SE-2*, 4 (Dec. 1976), 253–265.

Language Design for Reliable Software

S.L. Graham Editor

Abstraction Mechanisms in CLU

Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert
Massachusetts Institute of Technology

CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions – procedural, control, and especially data abstractions – are useful in the programming process. Of these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. CLU provides, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. This paper provides an introduction to the abstraction mechanisms in CLU. By means of programming examples, the utility of the three kinds of abstractions in program construction is illustrated, and it is shown how CLU programs may be written to use and implement abstractions. The CLU library, which permits incremental program development with complete type checking performed at compile time, is also discussed.

Key Words and Phrases: programming languages, data types, data abstractions, control abstractions, programming methodology, separate compilation
CR Categories: 4.0, 4.12, 4.20, 4.22

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant DCR74-21892.

A version of this paper was presented at the SIGPLAN/SIGOPS/SICSOFT Conference on Language Design for Reliable Software, Raleigh, N.C., March 28–30, 1977.

Authors' address: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.