ABSTRACTION and VERIFICATION in ALPHARD: Design and Verification of a Tree Handler

Mary Shaw
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

June, 1976

Abstract: The design of the Alphard programming language has been strongly influenced by ideas from the areas of programming methodology and formal program verification. The interaction of these ideas and their influence on Alphard are described by developing a nontrivial example, a program for manipulating the parse tree of an arithmetic expression.

Keywords and Phrases: abstraction and representation, abstract data types, assertions, correctness, information hiding, program specifications, program verification, programming languages, programming methodology, structured programming,

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 (Contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research).

Contents

Introduction

The major concerns of the Alphard research are the total cost of software development and the quality of the resulting programs. Problems that arise from repeated modifications to large programs, although often ignored in the literature, are of particular interest.

The Alphard language design has drawn heavily on previous work in both programming methodology and program verification. From the former we learned that in order to understand the programs we write, we must find some way to make them less complex; this may be done by restricting both the form of the programs (through modularity and localization of information [Parnas72]) and the process through which we create them (through stepwise refinement [Dijkstra72, Wirth71]). From the latter we learned that a programmer needs a precise, correct description of what a program does in order to use it without having to understand its implementation in detail; we also found techniques for writing and proving such descriptions.

Our concern with modifiability implies that the things we do to reduce program complexity must remain visibly part of the program. Thus it is not sufficient to develop a program in a well-structured fashion; the structure that was imposed must be obvious in the resulting program. The concept of *abstract data type* has therefore become central. In Alphard the concept is realized through a language mechanism called a form. The form is derived from the Simula class [Dahl72] in much the same way as the CLU cluster [Liskov74], and has the property that a programmer may reveal the behavior of some data type[1] to other users while concealing details of the implementation.

This explicit distinction between the abstract behavior of a data type and the concrete program which happens to implement that behavior provides an ideal setting in which to apply Hoare's techniques for proving data representations correct [Hoare72]. In the Alphard adaptation, we show (a) that the concrete representation is adequate to represent the abstract type, (b) that it is initialized properly, and (c) that each operator provided for the type both preserves the integrity of the representation and does what it is claimed to do (in terms of the abstract behavior and of the concrete procedure that happens to implement the operator). The specific formulas that must be proved are given below, and the methodology is discussed in [Wulf76].

This paper describes the language and verification methodology that have resulted from merging these ideas. A particular example is used to motivate the description, and a nonstandard implementation of the central data abstraction was chosen to emphasize the independence of the abstract and concrete definitions. The next section presents a problem

---

[1] In this paper we will use the word "type" in a nontechnical sense. In general, the abstraction introduced by a form need not be a type as we traditionally understand the word.

for which a binary tree is a natural primitive data structure; the specifications and procedures for the solution assume the existence of an implementation of binary trees.

The Alphard form which defines those binary trees is developed in the third section. The development of that form is essentially independent of the motivating example, so the resulting abstraction is useful for other applications as well.

Example: Minimal-Register Evaluation Order

Suppose you are given an arithmetic expression represented as a binary parse tree and you are asked to output the nodes in postfix form with the subexpressions arranged in the order that minimizes the number of registers required for the expression evaluation. An algorithm for finding this order was given by Nakata; its description was refined by Johnsson [Nakata67, Johnsson75]. The algorithm has two steps:

> Assign a weight W to each node n of the tree such that if n is a leaf then W_n=0, otherwise the immediate descendants of n have labels *right* and *left* and W_n = min(max(*left*+1, *right*), max(*left*, *right*+1)). W_n is the number of registers needed to evaluate the tree with root n.
>
> To evaluate the expression, begin at the root node and walk through the tree generating code so that at each node the operand requiring the larger number of registers is evaluated first. If the operands require the same number of registers, the left operand is evaluated first. If the right operand is evaluated first, include an indication of the reversal in the output stream.

Assuming that suitable definitions for trees and an output stream exist, this is easily converted to a program. We will use a data abstraction called a *btree* as if it were a primitive data type. It acts like a binary tree with an associated collection of node references called *bnodes*. There are at least enough operators on bnodes to obtain the left son, the right son, and the value field (nodeval) of any node and to determine whether a node is a leaf. The btree form given in Appendix A provides other operators, but they are not required for this example. For convenience, we restrict the size of btrees. We will use a queue to construct the output; a suitable definition is given in [Wulf76].

We first write, more precisely than the English algorithm above, an expression that describes the desired output for a parse tree E. This expression appears as the post conditon (output assertion) of the procedure *minreg* that computes it. We let W_{left} denote the weight of the left subtree, W_{right} denote the weight of the right subtree, and *invertop* supply the operator that indicates subexpression reversal.[2] The operator "~" denotes concatenation. The

two-step algorithm is then written:

```
minreg(E: btree(?r: record(wt,data: integer), ?maxht: integer)) returns P: queue
      post (isleaf(E) ⊃ minreg(E) = E.nodeval.data)
      ∧ (W_left ≥ W_right ⊃ minreg(E) = minreg(E.leftson)~minreg(E.rightson)~E.nodeval.data)
      ∧ (W_left < W_right ⊃ minreg(E) = minreg(E.rightson)~minreg(E.leftson)
          ~invertop~E.nodeval.data) =
  begin local exptr: bnode(E);
  markweights(exptr);
  minregwalk(exptr,P);
  end;
```

The program *minreg* operates on an arithmetic expression stored as a btree named E with a two-field record at each node and a known maximum height. The question marks on the btree parameters r and maxht indicate that those are implicit parameters -- that is, they will automatically be available for any btree which is passed as input. The record field names must, however, be exactly "data" and "wt". Minreg produces a queue named P from the tree E by first declaring a bnode variable, exptr, to point at nodes in E (exptr is automatically initialized to the root of E), then evaluating the register requirements of the subtrees with function *markweights*, and finally producing the queue with a special treewalk, *minregwalk*. Note that P is automatically initialized to the empty queue when the output variable for the procedure is set up.

Using M_k to denote the result of executing markweights on the tree with root k (e.g., M_{right} = markweights(exp.rightson)), we can write the definition of procedure markweights:

```
markweights(exp: bnode(?E: btree(?r: record(wt,data: integer), ?maxht: integer)))
      returns thiswt: integer
      post exp.nodeval.wt = M_exp ∧ (isleaf(exp) ⊃ M_exp = 0)
      ∧ (¬isleaf(exp) ⊃ M_n = min(max(M_left+1,M_right), max(M_left,M_right+1))) =
  begin local leftwt, rightwt: integer;
  if isleaf(exp) then thiswt ← 0
      else begin
      leftwt ← markweights(exp.leftson);
      rightwt ← markweights(exp.rightson);
      thiswt ← min(max(leftwt+1,rightwt), max(leftwt,rightwt+1));
      end;
  exp.nodeval.wt ← thiswt;
  end;
```

---

[2] In some cases we use qualified names rather than functional notation for clarity. Both styles are acceptable in Alphard, and no deep significance should be read into the

Markweights walks over exp (a bnode which indicates a subtree), setting the *wt* field at each node to the value described by the algorithm above. The post condition, located after the procedure header, specifies the result of the function. It is the formal description of what must be verified about procedure markweights and consequently a theorem about the use of that procedure. The body of markweight uses bnode functions named *isleaf*, *leftson*, *rightson*, and *nodeval*. It also refers to the *wt* field of the record stored as the value at each node. These operations are discussed in detail in the next section.

```
minregwalk(exp: bnode(?E:btree(?r:record(wt,data:integer),?maxht:integer)), order:queue)
        post (isleaf(exp) ⊃ order = order' ~ exp.nodeval.data)
        ∧ (W_left ≥ W_right ⊃ order = order' ~ Q_L ~ Q_R ~ exp.nodeval.data)
        ∧ (W_left < W_right ⊃ order = order' ~ Q_R ~ Q_L ~ invertop ~ exp.nodeval.data)
            where Q_L, Q_R are values which satisfy the post conditions of
                minregwalk(W_left,<>), minregwalk(W_right,<>) respectively =
begin
if ¬isleaf(exp) then
    if exp.leftson.nodeval.wt ≥ exp.rightson.nodeval.wt
        then begin minregwalk(exp.leftson,order); minregwalk(exp.rightson,order) end
        else begin minregwalk(exp.rightson,order);
            minregwalk(exp.leftson,order); enq(order,invertop) end;
enq(order,exp.nodeval.data);
end;
```

Minregwalk concatenates a postfix representation of its first argument (a parse tree) to its second argument (a queue). It tests the weights previously stored at the nodes in order to determine the evaluation order of the subtrees. The program uses the same functions on bnodes as markweights; it also uses a queue, but only performs the *enq* (enqueue) operation.[3] The formal definition and verification of queues is given elsewhere [Wulf76]; the usage in minregwalk should be clear.

Given suitable specifications of the functions on bnodes and queues, these two procedures can be shown to satisfy their post conditions.[4] The post conditions are, in turn, direct expressions of the algorithms given in English. It is straightforward, but neither necessary nor appropriate, to demonstrate that the post conditions express the minimal-register property. The algorithms themselves were acceptable on the strength of the analysis that accompanied them, and nothing would be gained by repeating that analysis for the formulation in the program.

---

[3] The *enq* function appends its second argument to the queue named by the first argument (i.e., enq(Q,e) = Q *append* e). The queue was created (initially empty) in the top-level procedure minreg for the purpose of collecting the output.

[4] The detailed proofs are standard and would contribute little to this exposition of Alphard.

In the next section we define, implement, and verify btrees and their associated bnodes, showing how the information needed to understand their behavior is kept separate from the information about their implementation.

Definition and Verification of a Form

Alphard's data abstraction mechanism is the form, a syntactic device for encapsulating a set of data declarations, function definitions, and other information about implementation details while revealing to the user only selected information about the behavior of the abstraction. The verification shows that the implementation supports the behavior described in the specification. The programs in the previous section used "btree" and "bnode" in the same way that other languages use type names: we said that exp was a bnode and assumed that we could therefore perform certain operations on it. In this section we develop the form that defines btrees and bnodes. The definition includes not only the functions actually used by the procedures above, but also enough others to round out the form as a useful abstraction. For example, the form defines functions that might be used to construct the parse tree that minreg manipulates.

A form contains three major components. These are the specifications, which provide information to the user about the abstract behavior of the objects being defined, the representation, which defines the concrete data structures used to maintain the objects and which states certain of their properties, and the implementation, which contains the bodies of the operators. Thus the skeleton of the btree form is:

```
form btree(N: record, maxht: integer) =
  beginform
  specifications
     . . .
  representation
     . . .
  implementation
     . . .
  endform
```

where ellipses are used to denote text which will be filled in later. This form actually describes a variety of specific trees: both the maximum height of the btree, *maxht*, and the record to be stored at each node, *N*, are parameters to the instantiation of the form. Note that bnodes have also been treated as "types". One of the components of the btree form is the definition of bnode, which is a form in its own right. We will examine each of the components in turn; the fragments discussed here are assembled as a complete form definition in Appendix A.

Specifications of btree

The btree specifications explain what a btree is and how it can be used. They give the restrictions on the instantiation parameters (*requires*), say that a btree is a special kind of graph[5] (*let*, *invariant*, *initially*), list the operations that can be performed on it (*functions*), and give the specifications for bnodes which refer to a given btree (*form*).

```
specifications
    requires maxht ≥ 0
    let btree = <r:N, g:graph>
        where g = <nodes: {tr:N}, links: {<tr_i:N, w:boolean, tr_j:N>}>;
    invariant
        (<n,w,k_1>,<n,w,k_2>∈links ⊃ k_1 = k_2) ∧        ! unique left & right sons
        (<n,w,x> ∈ links ⊃ ∃y <n,1-w,y> ∈ links)          ! either zero or two sons
        ∀n ∈ nodes (<n,w,r> ¬∈ links                      ! r is the root
            ∧ pathcnt(r,n) = 1                           ! singly connected
            ∧ length(<r..n>) ≤ maxht)                    ! limited height
    initially btree = <r, <{r},{ }>>;
    functions
        root(tr:btree) returns res: bnode post res = r,
        height(tr:btree) returns h:integer post h = max_k st k=length(<r, . . ., x>)
                st (isleaf(x) ∧ root(r)),
```

The requires simply says that only nonnegative values of maxht (the maximum height of the tree) make sense. The let declares that a btree may be regarded as a distinguished root and a graph, and that graph concepts will be used to explain them. Since a graph consists of a pair of sets, the let goes on to describe these sets in terms of booleans and the record type passed as an instantiation parameter. The invariant states certain relations on the graph which must *always* hold of a btree; the comments (! . . .) give the intuitive interpretation of each phrase. Initially states that when a btree is originally instantiated, it is empty except for the root. For each function, the specifications give the function name, its input parameters, its result (if any), and the abstract pre and post conditions needed for verifying the function and describing its inputs and outputs. The invariant will always be implicitly *and*ed with these explicit clauses to give the actual pre and post conditions. The functions root and height are applicable to any btree (i.e., any one for which the invariant holds), so the constant true as an explicit pre condition is omitted.

Finally, the btree specifications give the abstract description of the sub-form bnode. The latter form's organization is similar to btree's, except that the specifications of bnode have been printed with those of btree in order to localize the information that will be presented to a user.

---

5 A suitable definition of graphs is given in Appendix B.

```
form bnode(T:btree(?N:record,maxht:integer)) =
    beginform
    specifications
    let bnode = ptr:N;
    invariant ptr ∈ nodes;
    initially ptr = r;
    functions
        leftson(tr:bnode) returns subtr:bnode
            pre ¬isleaf(tr) post <tr,0,subtr> ∈ links,
        rightson(tr:bnode) returns subtr:bnode
            pre ¬isleaf(tr) post <tr,1,subtr> ∈ links,
        isleaf(tr:bnode) returns tv:boolean
            post tv ≡ ∀w ¬∃subtr <tr,w,subtr> ∈ links,
        isroot(tr:bnode) returns tv:boolean
            post tv ≡ ∀w ¬∃subtr <subtr,w,tr> ∈ links,
        father(tr:bnode) returns subtr:bnode
            pre ¬root(tr) post ∃w st <tr,w,subtr> ∈ links;
        ancestor(tr,subtr:bnode) returns tv:boolean
            post tv ≡ tr=subtr ∨ ∃p=<tr, . . ., subtr> st path(p),
        extend(tr:bnode) pre isleaf(tr) ∧ height(tr) < maxht
            post ¬isleaf(tr) ∧ isleaf(rightson(tr)) ∧ isleaf(leftson(tr))
    selectors
        nodeval: N;
    endform
```

The post conditions of leftson and rightson indicate that a weight of 0 on an arc denotes a left son, while a weight of 1 denotes a right son. The only thing new here is the selectors, which may be viewed as field-accessors. A name declared as a selector may be used both to set and to fetch values. Note that a bnode is always associated with a *particular* btree.

Representation of btree

The representation part shows how btrees are actually stored in terms of other data structures (*unique*, *invariant*) and explains the correspondence between this concrete representation and the abstract description given in the specifications (*rep*).

```
representation
    unique T: vector(rec: record(node:N, inuse:boolean),1,2^(maxht+1)-1)
        init begin for x:invec(T) do x.inuse ← false; T[1] ← rec(null,true) end;
    rep(T) = < T[1].node, < {T[i].node | T[i].inuse}, {<T[i].node,w,T[2i+w].node>
            | T[i].inuse ∧ T[2i+w].inuse ∧ w∈{0,1} } > >;
    invariant .T[1].inuse  ∧ (T[i].inuse ⊃ i=1 ∨ T[i div 2].inuse∧T[i+1-2(i mod 2)].inuse);
```

The unique declaration states that each btree will consist of a vector of records (node value and "inuse" bit) indexed from 1 to $2^{maxht+1}-1$. Alphard's scope rules prevent the vector and the record field names from being used outside the form. The init clause of the declaration gives the initialization code to be executed when that vector is allocated.[6] It sets all inuse bits to false, then sets the record at the root to (null,true). The unique declaration states that each instance of a btree will get its own vector.

The terms rep(T) and invariant explain how the vector is interpreted as a representation of an abstract tree. The representation function rep(T) exhibits an ordered pair consisting of the node field of T[1], which represents the root, and a pair of sets which represent the graph. The invariant gives a restriction on the distribution of inuse bits which is sufficient to enforce the abstract invariant.

In the representation chosen for this version of btree, all nodes are stored in a vector and the j^{th} node's sons are found at positions 2j and 2j+1. The inuse bit distinguishes whether potential tree positions are actually included in the tree; a separate bit was set aside for this purpose because the node can be an arbitrary record and, as a result, there is no way to encode "nonexistence" in the node value itself. Note that this is the first time a specific implementation strategy has been mentioned: up to this point a linked-list strategy should have seemed equally plausible.

Verification Considerations

We turn now to the question of how we decide whether a form will actually behave as promised by its abstract specifications -- that is, what properties of a form must be verified if we wish to use its instantiations with confidence. 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[Hoare72].

The abstract object and its behavior are described in terms of some mathematical entities natural to the problem domain. Graphs are used here to describe btrees; sequences are used in [Wulf76] to describe queues and stacks, and so on. In btree we appeal to graphs:

- in the invariant, which explains that a btree is a graph that meets certain restrictions,

- in the initially clause, where a particular graph and its root are displayed, and

---

[6] The phrase "for x: invec(T)" invokes the Alphard iteration statement for vectors. It causes the loop to be executed once for each element in the vector. See [Shaw76] for further discussion of iteration.

- in the pre and post conditions for each function, which describe the effect the function has on a graph which satisfies the invariant.

The form contains a parallel set of descriptions of the concrete object and how it behaves. Since btrees are implemented in terms of a vector of records, the concrete specifications give restrictions and effects on that vector. 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 rep(T), which gives a mapping from a vector of records to a graph and its root. The purpose of a form verification is to ensure that the two invariants and the rep(T) 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[7]:

For the form

1. Representation validity
   $I_c(T) \supset I_a(rep(T))$

2. Initialization
   requires { *init clause* } initially(rep(T)) $\wedge$ $I_c(T)$

For each function

3. Concrete operation
   in(T) $\wedge$ $I_c(T)$ { *function body* } out(T) $\wedge$ $I_c(T)$

4. Relation between abstract and concrete
   4a. $I_c(T) \wedge pre(rep(T)) \supset in(T)$
   4b. $I_c(T) \wedge pre(rep(T')) \wedge out(T) \supset post(rep(T))$

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

[7] We will use $I_a(rep(T))$ to denote the abstract invariant of an object whose concrete representation is T, $I_c(T)$ 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(T'))" refers to the value of T *before* execution of the function. A complete development of the form verification methodology appears in [Wulf76].

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.

For btree, several of these steps will be simplified by appealing to the following *standard construction*, which determines the correspondence between an index in the vector representation and a path from the root to a node in the abstract graph.

Let T[j] be the vector element which represents some node in a btree.
Let $w_0 w_1 w_2 \ldots w_k$ be the binary representation of j, $w_0=1$.
Define p_i as $p_i = SUM_{b=0\ldots i} (w_b 2^{i-b})$ for i=0...k (note that $p_i=2p_{i-1}+w_i$ and $p_k=j$).
Then the (abstract) path from the root to a node is the path whose elements are
$<T[p_{i-1}].node, w_i, T[p_i].node>$ for i = 1...k

In addition, if the node is in the tree, T[j].inuse = true and, because of the term
T[i].inuse ⊃ i=1 ∨ T[i *div* 2].inuse
of the invariant, all elements in the path are also in the tree.

Verification of form properties of btree

At this point we have enough information about btrees to perform verification steps 1 and 2, which show the overall validity of the form. We can now proceed with an informal proof of these steps.

1. Representation Validity
   Show: T[1].inuse ∧ (T[i].inuse ⊃ i=1 ∨ T[i *div* 2].inuse∧T[i+1-2(i *mod* 2)].inuse) ⊃
   $(<n,w,k_1>,<n,w,k_2> \in$ links ⊃ $k_1=k_2$) ∧ ($<n,w,x> \in$ links ⊃ ∃y $<n,1-w,y> \in$ links)
   ∀n ∈ nodes ($<n,w,r> \notin$ links ∧ pathcnt(r,n)=1 ∧ length(<r...n>) ≤ maxht)
   where nodes = {T[i].node | T[i].inuse}
   links = {<T[i].node,w,T[2i+w].node> | T[i].inuse∧T[2i+w].inuse ∧ w∈{0,1}}
   Proof: Take the clauses of the conclusion one by one:
   (a) $k_1=k_2$ because the rep function uniquely determines the triples in links on the basis of n and w.
   (b) ∃y $<n,1-w,y>$ because both sons or neither son of a node have the inuse bit set.
   (c) $<n,w,r> \notin$ links because r=T[1].node and $1 \neq 2i+w$ for any integer $i \geq 1$.
   (d) pathcnt(r,n) = 1 because the standard construction is unique.
   (e) length(<r...n>) ≤ maxht because each vector index must be in the range $[1..2^{maxht+1}-1]$ and the standard construction gives a path whose length is the number of significant bits in the vector index.

2. Initialization

Show: maxht ≥ 0 { for x:invec(T) do x.inuse ← false; T[1] ← rec(null,true) }
btree = <T[1].node, <{T[1].node}, {}>> ∧ T[1].inuse
∧ (T[i].inuse ⊃ (i=1 ∨ T[i *div* 2].inuse∧T[i+1-2(i *mod* 2)].inuse))

Proof: We will pass over the verification of the for loop; it sets all inuse bits to false (see [Shaw76] for details). The uniterated assignment complete the initialization by making T[1] the only active node.

These steps demonstrate that any vector T which satisfies I_c represents a legal limited-height btree and that the initial value of a newly-instantiated btree is initialized properly. We will show below that each function preserves the accuracy of the representation, but the adequacy of that representation is established here.

Implementation of btree

The implementation part gives the bodies of the two functions and the bnode form promised by the specifications. For each function, we provide both the program to compute the function and the concrete in and out conditions. Although neither function is used in the minreg program, they are included in the btree form in order to make it a more generally useful abstraction. The verification of these functions is omitted here because the technique is illustrated below for functions we have actually used.

```
implementation
    body root out res = 1 =
        ! The bnode return parameter is initialized to the root.

    body height out h=log(max_i st T[i].inuse) =
        first j: downto(2^maxht+1-1,1) suchthat T[j].inuse
            then h ← floor(log_2 j);
```

Implementation of bnode

The bnode form is organized like the btree form, and its verification proceeds in a similar fashion. Its specifications were given as part of the btree specifications. We now look at its representation, which is simply an integer index into the vector which represents the btree:

```
representation
    unique ptr: integer init ptr ← 1;
    rep(ptr) = T[ptr].node;
    invariant 1≤ptr≤2^maxht+1-1 ∧ T[ptr].inuse;
```

To verify the form properties, we must prove two things:

1. Representation validity
   Show: $1 \le ptr \le 2^{maxht+1}-1 \wedge T[ptr].inuse \supset T[ptr].node \in \{ T[i].node \mid T[i].inuse \}$
   Proof: Clear.

2. Initialization
   Show: true { ptr←1 } $T[ptr].node = T[1].node \wedge 1 \le ptr \le 2^{maxht+1}-1$
   $\wedge$ T[ptr].inuse
   Proof: Applying the rep function and the assignment axiom, this becomes
   $T[1].node = T[1].node \wedge 1 \le 1 \le 2^{maxht+1}-1 \wedge T[1].inuse$
   This reduces to T[1].inuse, which is assured by the concrete invariant of btree.

Thus we have shown that the representation supports the abstraction. We will next discuss and verify some of the functions used by the programs of the previous section. Other functions are given in the form definition in Appendix A. Note that the invariants of btree (as well as those of bnode) must be preserved. This step is omitted from the proofs given here because no part of the btree representation is altered.

One of the simplest functions finds the *left son* of a given node. Its abstract specifications and body are:

```
leftson(tr:bnode) returns subtr:bnode
    pre ¬isleaf(tr) post <tr,0,subtr> ∈ links
. . .
body leftson in ¬isleaf(tr) out subtr.ptr = 2*tr.ptr =
    subtr.ptr ← 2*tr.ptr;
```

The program itself is clear: double a node's index to find its left son. The in condition asserts that the leftson function may not be applied to a leaf[8] The out condition repeats the doubling property. Recall that the concrete invariant must be shown to hold along with the in and out conditions, so we may be sure leftson is applied only to legal bnodes and does not destroy them. These properties are verified formally by proving the following (again I_c denotes the concrete invariant):

[8] This design decision forces the user to extend the tree explicitly before using new nodes, but it offers a degree of protection against errors that automatic tree growth would not. We could, of course, extend the tree automatically when leftson or rightson is applied to a leaf, but that is a different decision and leads to a different program.

3. Concrete operation

Show: $\exists j$ (j *div* 2 = tr.ptr $\wedge$ tr.T[j].inuse $\wedge$ $1 \le j \le 2^{maxht+1}-1$) $\wedge$ I_c

{ subtr.ptr ← 2*tr.ptr } subtr.ptr = 2*tr.ptr $\wedge$ I_c

Proof: Choosing j=2*tr.ptr and applying the assignment axiom, we obtain

tr.ptr = tr.ptr $\wedge$ tr.T[2*tr.ptr].inuse $\wedge$ $1 \le 2*tr.ptr \le 2^{maxht+1}-1$ $\wedge$ I_c

$\supset$ 2*tr.ptr=2*tr.ptr $\wedge$ I_c

The concrete invariant for tr is maintained since tr is not modified; it is established for subtr because of the range check on j.

4a. in condition holds

Show: $1 \le tr.ptr \le 2^{maxht+1}-1$ $\wedge$ tr.T[tr.ptr].inuse $\wedge$ $\exists w,s$ <tr,w,s>$\in$links

$\supset$ $\exists j$ (j *div* 2=tr.ptr $\wedge$ tr.T[j].inuse $\wedge$ $1 \le j \le 2^{maxht+1}-1$)

Proof: If $\exists w,s$ *st* <tr,w,s> $\in$ links, then s must correspond to the vector element indexed by 2*tr.ptr+w, and it must be an active node. This is sufficient to establish the conclusion.

4b. post condition holds

Show: $1 \le tr.ptr \le 2^{maxht+1}-1$ $\wedge$ tr.T[tr.ptr].inuse $\wedge$ $\exists w,s$ <tr,w,s> $\in$ links

$\wedge$ subtr.ptr=2*tr.ptr $\supset$ <tr,0,s> $\in$ links

Proof: The concrete invariant of btree says that the s must be 2*tr.ptr+w and that both <tr,0,s> and <tr,1,s> exist, which is precisely the condition needed.

Note that the proof refers to both the ptr field of the input parameter tr and the tree T for which tr was created. Qualified names may be used for this selection, so we write tr.ptr and tr.T, respectively. These phrases can be further qualified, so we can select a particular element of vector tr.T by writing tr.T[i] (since T is a vector of records) and the inuse field of that vector element by writing tr.T[i].inuse. The definition and verification of rightson are essentially the same.

We often needed to determine whether the tree we had in hand was a leaf. The specifications and function body for *isleaf* are

```
isleaf(tr:bnode) returns tv:boolean
    post tv ≡ ∀w ¬ ∃subtr <tr,w,subtr> ∈ links
. . .
body isleaf
    out tv ≡ (¬∃j (j div 2 = tr.ptr ∧ tr.T[j].inuse ∧ 1≤j≤2^(maxht+1)-1)) ≡
    tv ← tr.ptr > 2^maxht-1 ∨ (¬tr.T[2*tr.ptr].inuse ∧ ¬tr.T[2*tr.ptr+1].inuse);
```

The out condition specifies that isleaf returns "true" if there is no vector index in range for which T[j] is both in use and a left or right son of the input. Since the in condition is omitted, it is assumed to be identically true, so isleaf must be applicable to any btree. To verify isleaf, we must show the following:

3. Concrete operation
Show: I_c { tv ← tr.ptr > $2^{maxht-1}$ ∨ (¬tr.T[2*tr.ptr].inuse ∧ ¬tr.T[2*tr.ptr+1].inuse) }
(tv ≡ ¬∃j (j *div* 2 = tr.ptr ∧ tr.T[j].inuse ∧ 1≤j≤$2^{maxht+1-1}$].inuse)) ∧ I_c
Proof: Rewriting to eliminate j and applying the assignment axiom, this becomes

I_c ⊃ [(tr.ptr > $2^{maxht-1}$ ∨ (¬tr.T[2*tr.ptr].inuse ∧ ¬tr.T[2*tr.ptr+1].inuse)
≡ ¬((tr.T[2*tr.ptr].inuse ∨ tr.T[2*tr.ptr+1].inuse) ∧ (1≤2*tr.ptr≤$2^{maxht+1-1}$
∨ 1≤2*tr.ptr+1≤$2^{maxht+1-1}$)) ∧ I_c)]

which in turn reduces to

I_c ⊃ [(tr.ptr > $2^{maxht-1}$ ∨ (¬tr.T[2*tr.ptr].inuse ∧ ¬tr.T[2*tr.ptr+1].inuse)
≡ ¬((tr.T[2*tr.ptr].inuse ∨ tr.T[2*tr.ptr+1].inuse) ∧ (1≤tr.ptr≤2^{maxht}
∨ 1≤tr.ptr+1≤2^{maxht}) ∧ I_c)]

which is clear.

4a. in condition holds
Show: I_c ⊃ true
Proof: Clear.

4b. post condition holds
Show: I_c ∧ tv ≡ ¬∃j (j *div* 2 = tr.ptr ∧ tr.T[j].inuse ∧ 1≤j≤$2^{maxht+1-1}$].inuse)
⊃ ∀w ¬∃subtr(<tr,w,subtr> ∈ links)
Proof: The out says there is no w for which T[t*tr.ptr+w].inuse, either because 2*tr.ptr would exceed the index range of the array or because the inuse bit is set to false. By the definition of links, there is no triple <tr.T[tr.ptr],w,tr.T[2*tr.ptr+w]> which could correspondto <tr,w,subtr>.

Finally, bnode provides a selector, nodeval, for performing fetches and stores to the value field of a tree node. The implementation of nodeval is given by

map nodeval = T[ptr].node;

Changing this particular field has no effect on any invariant, so nothing must be proved.

Conclusion

This paper has used a concrete example to explain the Alphard philosophy on the development and verification of programs. The example was nontrivial; it implemented the abstraction with a nonstandard representation, and it involved a subtype. Several aspects of the development deserve special notice.

First, note that we did *not* verify the "main program". The program was simply a restatement of an algorithm that had undergone considerable analysis in another formulation. It would have been unreasonable to redo that analysis in the course of verifying the program. We therefore indicated that it was sufficient to ensure that the program was an accurate restatement of the algorithm. If program verification is ever to impact real programs, we must take such steps to avoid reproving all programs from first principles. Since the form encapsulates a collection of related information about how some abstract behavior is to be achieved, it is a reasonable body of information about which to prove theorems. This is evidenced by the nearly complete independence of the discussions of the minreg program and the btree form.

Next, the form presented in Appendix A contains functions not actually used by the program of the example. We believe that in the future libraries of forms will develop, and that these will be more useful than present libraries because the forms are verified and because verification considerations stimulated careful thought about what constitutes a good abstraction. Further, the explicit distinction between the abstract specification and the concrete implementation should simplify modification of the code both because the assumptions on which users depend are made clear and also because only part of the verification should have to be repeated.

Finally, some of our colleagues have expressed concern over the length of Alphard programs. Certainly the verification information adds text, but we believe that this information must be supplied somewhere. Nakata gave an Algol program for converting a parse tree to code [Nakata67]. That program performs a slightly different operation from minreg, so an exact comparison is impossible, but if we ignore verification information and the btree functions that were never used, the number of lexemes in the Alphard procedures and forms is within 10% of the number of lexemes in Nakata's program. This crude comparison supports our feeling that the program text itself is not excessively large.

Acknowledgements

The abstractions, programs, and verifications presented here have benefited from comments by a number of my colleagues in the Alphard project, particularly Bill Wulf and Ralph London.

Appendix A
Complete Definition of Btree and Bnode

```
form btree(N:record, maxht:integer) =
  beginform
  specifications
    requires maxht ≥ 0
    let btree = <r:N, g:graph>
        where g = <nodes: {tr:N}, links: {<tr_i:N, w:boolean, tr_j:N>}>;
    invariant
        (<n,w,k_1>,<n,w,k_2>∈links ⊃ k_1 = k_2) ∧          ! unique left & right sons
        (<n,w,x> ∈ links ⊃ ∃y <n,1-w,y> ∈ links)           ! either zero or two sons
        ∀n ∈ nodes (<n,w,r> ¬∈ links                        ! r is the root
            ∧ pathcnt(r,n) = 1                             ! singly connected
            ∧ length(<r..n>) ≤ maxht)                      ! limited height
    initially btree = <r, <{r},{ } >>;
    functions
        root(tr:btree) returns res: bnode post res = r,
        height(tr:btree) returns h:integer post h = max_k st k=length(<r, . . ., x>)
                st (isleaf(x) ∧ root(r)),
    form bnode(T:btree(?N:record,maxht:integer)) =
        beginform
        specifications
        let bnode = ptr:N;
        invariant ptr ∈ nodes;
        initially ptr = r;
        functions
            leftson(tr:bnode) returns subtr:bnode
                pre ¬isleaf(tr) post <tr,0,subtr> ∈ links,
            rightson(tr:bnode) returns subtr:bnode
                pre ¬isleaf(tr) post <tr,1,subtr> ∈ links,
            isleaf(tr:bnode) returns tv:boolean
                post tv ≡ ∀w ¬∃subtr <tr,w,subtr> ∈ links,
            isroot(tr:bnode) returns tv:boolean
                post tv ≡ ∀w ¬∃subtr <subtr,w,tr> ∈ links,
            father(tr:bnode) returns subtr:bnode
                pre ¬root(tr) post ∃w st <tr,w,subtr> ∈ links;
            ancestor(tr,subtr:bnode) returns tv:boolean
                post tv ≡ tr=subtr ∨ ∃p=<tr, . . ., subtr> st path(p),
            extend(tr:bnode) pre isleaf(tr) ∧ height(tr) < maxht
                post ¬isleaf(tr) ∧ isleaf(rightson(tr)) ∧ isleaf(leftson(tr))
```

```
        selectors
            nodeval: N;
        endform

    representation
        unique T: vector(rec: record(node:N, inuse:boolean),1,2^(maxht+1)-1)
            init begin for x:invec(T) do x.inuse ← false; T[1] ← rec(null,true) end;
        rep(T) = < T[1].node, < {T[i].node | T[i].inuse}, {<T[i].node,w,T[2i+w].node>
                | T[i].inuse ∧ T[2i+w].inuse ∧ w∈{0,1} } > >;
        invariant T[1].inuse  ∧ (T[i].inuse ⊃ i=1 ∨ T[i div 2].inuse∧T[i+1-2(i mod 2)].inuse);

    implementation
        body root out res = 1 =
            ! The bnode return parameter is initialized to the root.

        body height out h=log(max_i st T[i].inuse) =
            first j: downto(2^(maxht+1)-1,1) suchthat T[j].inuse
                then h ← floor(log_2 j);

        formbody bnode =
        beginform
        representation
            unique ptr: integer init ptr ← 1;
            rep(ptr) = T[ptr].node;
            invariant 1≤ptr≤2^(maxht+1)-1 ∧ T[ptr].inuse;

        implementation
        body leftson in ¬isleaf(tr) out subtr.ptr = 2*tr.ptr =
            subtr.ptr ← 2*tr.ptr;

        body rightson in ¬isleaf(tr) out subtr.ptr = 2*tr.ptr+1 =
            subtr.ptr ← 2*tr.ptr + 1;

        body isleaf
            out tv ≡ (¬∃j (j div 2 = tr.ptr ∧ tr.T[j].inuse ∧ 1≤j≤2^(maxht+1)-1)) =
            tv ← tr.ptr > 2^maxht-1 ∨ (¬tr.T[2*tr.ptr].inuse ∧ ¬tr.T[2*tr.ptr+1].inuse);

        body isroot out tv ≡ (tr.ptr=1) =
            tv ← tr.ptr=1;

        body father in tr.ptr > 1 out subtr.ptr = tr.ptr div 2 =
            subtr.ptr ← tr.ptr div 2;
```

```
body ancestor in tr.T[tr.ptr].inuse ∧ tr.T[subtr.ptr].inuse
        out tv ≡ (∃j1,j2,...jK st j1=tr.ptr ∧ jK=subtr.ptr
        ∧ tr.T[ji].inuse ∧ ji-1=ji div 2) =
    begin local shftd;
    shftd ← floor(log2 subtr.ptr) - floor(log2 tr.ptr);
    tv ← tr.ptr = subtr.ptr div 2^shftd; end;

body extend in isleaf(tr.ptr) ∧ tr.ptr<2^maxht
        out ¬isleaf(tr.ptr) ∧ isleaf(rightson(tr.ptr)) ∧ isleaf(leftson(tr.ptr)) =
    begin
    tr.T[2*tr.ptr].inuse ← tr.T[2*tr.ptr+1].inuse ← true;
    tr.T[2*tr.ptr].node ← tr.T[2*tr.ptr+1].node ← null;
    end;

map nodeval = T[ptr].node;
endform;

endform;
```

Appendix B
Formal Definition of Graphs with Weighted Arcs

This formal definitiion is based on the definition of *graph* given by Knuth [Knuth73, sec. 2.3.4] with the addition of labels, or *weights*, on the arcs.

1. Let N be a set called the *node domain* of a graph and let W be a set called the *arc weights* of a graph.

   (a) An *arc* is a triple $\langle n_i,w_j,n_k\rangle$ where $n_i \in N$, $w_j \in W$, $n_k \in N$
   (b) A *graph* is a pair $\langle E,A\rangle$ where E is a set of nodes and A is a set of arcs such that $\langle n_i,w_j,n_k\rangle \in A \supset n_i,n_k \in E$
   (c) These are the only graphs.

2. The notation $\langle n_1,n_2,\ldots,n_k\rangle$ is an abbreviation for $\{ \langle n_1,w_1,n_2\rangle, \langle n_2,w_2,n_3\rangle, \ldots, \langle n_{k-1},w_{k-1},n_k\rangle \}$ for any values of w_j

3. The following functions and relations are defined for $G = \langle E,A\rangle$ and $n_i \in E$:

   (a) $adj(n_1,n_2) \equiv_{df} \exists w \; st \; \langle n_1,w,n_2\rangle \in A$
   (b) $pathcnt(n_1,n_2) \equiv_{df}$ cardinality($\langle n_1,\ldots,n_2\rangle \; st \; \langle n_i,w,n_{i+1}\rangle \in A, i \in [1..k-1]$)
   (c) $path(n_1,n_k) \equiv_{df} \langle n_1,\ldots,n_k\rangle \; st \; \langle n_i,w,n_{i+1}\rangle \in A, i \in [1..k-1]$
   (d) $simple(\langle n_1,n_2,\ldots,n_k\rangle) \equiv_{df} (n_i = n_j \supset \{i,j\} = \{1,k\}$
   $\wedge \; pathcnt(n_1,n_k)=1, i,j \in [1..k]$
   (e) $strngconn(G) \equiv_{df} \forall i,j \; path(i,j)$
   (f) $connected(g) \equiv_{df} strngconn(GX)$.
   where $GX = \langle G.E, G.A \cup \{\langle a,b,c\rangle \mid \langle c,b,a\rangle \in G.A \}\rangle$
   (g) $length(\langle n_1,n_2,\ldots,n_k\rangle) \equiv_{df} k$
   (h) $cycle(n_i,n_j) \equiv_{df} \exists \langle x_i,\ldots,x_j\rangle \; st \; simple(\langle x_i,\ldots,x_j\rangle)$
   $\wedge \; i=j \wedge length(\langle x_i,\ldots,x_j\rangle) \geq 3$
   (i) $G = H \equiv_{df} G.A = H.A \wedge G.E = H.E$

References

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

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

[Hoare72] C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica*, 1,4, 1972 (pp. 271-281).

[Johnsson75] Richard K. Johnsson, "An Approach to Global Register Allocaton", *Carnegie-Mellon University Technical Report*, December 1975.

[Knuth73] Donald E. Knuth, *Fundamental Algorithms*, Second Edition, Addison-Wesley, 1973.

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

[Nakata67] Ikuo Nakata, "A Note on Compiling Algorithms for Arithmetic Expressions", *Communications of the ACM*, 10, 8, August 1967.

[Parnas72] David L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15, 12, December 1972 (pp. 1053-1058).

[Shaw76] Mary Shaw, William Wulf, and Ralph L. London, "Abstraction and Verification in Alphard: Iteration and Generators", *Carnegie-Mellon University Technical Report* and *USC Information Sciences Institute Research Report*, 1976.

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

[Wulf76] William Wulf, Ralph L. London, and Mary Shaw, "Abstraction and Verification in Alphard: Introduction to Language and Methodology", *Carnegie-Mellon University Technical Report* and *USC Information Sciences Institute Research Report*, 1976.