Proceedings of the IEEE Conference on Specifications of Reliable Software, 1979, pp. 222-237. Reprinted in M. Shaw (Ed.), Alphard: Form and contect (pp. 255-281). New York, NY: Springer-Verlag, 1981.

An Alphard Specification of a Correct and Efficient Transformation on Data Structures

Jon Louis Bentley[1] and Mary Shaw

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

In this paper we study standard program components applicable to a wide variety of design tasks; we choose for this study the specific problem domain of *data structures for general searching problems.* Within this domain Bentley and Saxe [1978] have developed transformations for converting solutions of simple searching problems to solutions of more complex problems. We discuss one of those transformations, specify precisely the transformation and its conditions of applicability, and prove its correctness; we accomplish this by casting it in terms of abstract data types -- specifically by using the Alphard **form** mechanism. We also demonstrate that the costs of the structures derived by this transformation are only slightly greater than the costs of the original solutions. The transformation we describe has already been used to develop a number of new algorithms, and it represents a new level of generality in software engineering tools.

1. Introduction

In this paper we demonstrate the use of data abstraction techniques to specify precisely and verify a very general definition that yields solutions to a broad class of problems. Within a specific problem domain we explore ways to transform mechanically a solution of one kind of problem to obtain a solution of another kind of problem *without using detailed knowledge about the implementation of the original solution.* We investigate the utility of data abstraction techniques, particularly formal specifications, in defining the transform and the conditions under which it can be applied.

Our chosen problem domain is *data structures for general searching problems.* In a searching problem we must organize a set of objects so that queries about that set can be answered quickly. We elaborate this definition of searching problems and give several examples in Section 2.

Many (if not most) searching problems can be solved by simply scanning the data set in response to each query; the time to process such a query will then be (at least) proportional to the number of elements stored. We will be concerned exclusively with searching problems that require very efficient solutions; we must therefore store the set in a data structure that is more sophisticated than a simple sequence.

A particular searching problem may arise in either of two distinct forms. In the *static* case the structure is built once-and-for-all, and then all queries are asked. In the *dynamic* case the structure is initially empty and queries are interspersed with the insertion of new elements. Both of these forms arise in practice. A solution to a searching problem is given by a *data structure* and some algorithms for operating on the structure. In general it appears that the task of designing efficient data structures for dynamic problems is intrinsically more difficult than designing static data structures. This paper presents a formal description of a method due to Bentley and Saxe [1978] whereby a static data structure for a particular searching problem can be automatically *transformed* into a dynamic structure for the same problem.

We have chosen Alphard (see Hilfinger [1978]) for the formal statement of specifications and programs. This decision has several good effects. First, it gives us an expressive tool: the style imposed by the language encourages precise, formal statements of assumptions and calls attention to places where assumptions about types or values are being made. Second, the specification methodology is well-suited to high-level, abstract descriptions; this is particularly true for the transformation discussed here, because the transformation applies in a broad but well-defined set of circumstances. Third, Alphard provides for merging code and abstract specifications gracefully, which allows us to move smoothly from specification to implementation and to prove the correctness of the result.

This research was supported in part by the National Science Foundation under Grant MCS77-03883 and in part by the Office of Naval Research under Contract N00014-76-C-0370.

1. Also with the Department of Mathematics.

The technique demonstrated here makes it possible to write verified components that extend the *abstract* properties of other programs, and to do so independent of implementation details. The ability to specify the dependencies formally opens the doors to a new class of verified library entities. Not only can we specify the properties provided by a library unit, we can also specify the minimum properties that must be supplied to it by submodules, subprograms, data types, or other more primitive definitions.

Before we describe the general transformation we need a more complete definition of searching problems; this is provided in Section 2. Section 3 uses a particular example to illustrate and motivate the general transform. Section 4 then develops the general case, including a complete Alphard **form** and its verification. Conclusions are offered in Section 5.

2. Searching Problems

Many computing problems are stated directly as searching problems; others are reducible to searching problems. Perhaps the best-known example of a searching problem is *member searching*. In this problem we are given a set F of elements (the *stored elements*) to organize so that subsequent queries asking if a given element x (the *query object*) is in F can be answered quickly. Member searching arises in such applications as database systems and statistical packages; many other problems (such as symbol tables in compiler building) can be reduced to member searching. Knuth [1973] describes many data structures available for solving the member searching problem. To analyze a particular data structure S we give three functions of n (the number of elements in the set F) describing the cost of searching: P(n), the *processing time* required to organize the set into a data structure; Q(n), the time required to answer a *query*; and S(n), the *storage* required by the structure. The simplest structure for member searching is the "linear search" structure in which the elements are stored as a sequence and a search then compares the new element x to every element in F. This structure has costs P(n) = O(n), Q(n) = O(n), and S(n) = O(n). (All of the structures we will see throughout this paper have linear storage costs; we will often omit this cost for brevity.) A more sophisticated structure is the sorted array combined with binary search; its costs are P(n) = O(n lg n) and Q(n) = O(lg n). Thus if many searches are expected to be performed, it is cost-effective to organize the set in advance to decrease query costs.

Our discussion of member searching has so far been limited to the *static* case: all of the elements were presented for processing before any queries were handled. Many applications, however, demand a *dynamic* structure -- one in which new elements may be inserted as the queries are processed. To analyze the computational efficiency of a dynamic structure we give the same three cost functions as before: Processing, Query and Storage costs. In this context, however, P(n) denotes the *total time required to insert the first n elements*. There are many kinds of balanced tree data structures that perform dynamic member searching with costs P(n) = O(n lg n) and Q(n) = O(lg n) (see Knuth [1973]). It is pleasing to note that in making the transition from static to dynamic member searching, the asymptotic complexity of the algorithms does not increase. This apparently happy situation is marred by the realities of implementation, however: the constants hidden in the "big-ohs" of the cost functions are all substantially increased in the dynamic case, and the programs for manipulating balanced trees are much more complex than those for static structures.

There are many other types of searching problems besides member searching. In general, a (static) searching problem calls for organizing some set F of stored objects into a data structure S so that queries concerning a new query object x can be answered quickly. To illustrate the general problem we can consider a particular searching problem defined when F is a set of points in the plane and x is a new point in the plane (not necessarily in F): *nearest neighbor searching* calls for finding the point in F nearest to x. Another typical searching problem is *range searching*: F is again a set of points in the plane and x is a rectangle; the search must then list all points in F that lie in rectangle x. Many search structures are known for these and other static searching problems. Unfortunately, not much at all is known about efficient dynamic structures for these searching problems. We will soon see, however, a general transformation that allows a solution for a static searching problem to be converted into a solution for a dynamic problem, as long as the search problem satisfies a very weak condition.

3. A Transform for Nearest Neighbor Searching

In this section we will investigate the problem of nearest neighbor searching. This problem calls for organizing a set F of n points in the plane so that the distance from the nearest point[1] in F to a new query point can be answered quickly. This problem arises in a host of applications such as geographic data bases and statistics (including density estimation, classification, and clustering). Until quite recently, no guaranteed fast ways of performing nearest neighbor searching were known. After a flurry of research activity on the problem, Lipton and Tarjan [1977] gave an algorithm for static nearest neighbor searching with costs

$P(n) = O(n \lg n)$,
$Q(n) = O(\lg n)$, and
$S(n) = O(n)$.

This algorithm can be proved to be optimal under a fairly strong model of computation.

Although Lipton and Tarjan's structure gives us a fast method for static nearest neighbor searching, it does not appear that their method is at all amenable to modification for performing dynamic searches. There are, however, two simple ways to build a dynamic structure. The obvious way is to store the points in the structure as a sequence and then examine each point to answer a query; this method has constant insertion time (therefore linear cost for n insertions) and linear query time. (This method can be viewed as storing n static structures, each of size one.) A more sophisticated method calls for always maintaining a single Lipton-Tarjan structure (abbreviated *LTS*) and rebuilding it as each point is inserted. This scheme will maintain the logarithmic query time, but the total cost for inserting n elements will be $P(n) = O(n^2 \lg n)$. Thus we see that it is possible to achieve very fast insertion times *or* query times; we would like, however, a data structure that achieves both.

3.1. Specifications

In order to achieve both fast insertions and fast queries, we will define a more elaborate data structure in terms of the LTS. It is obvious that we will need operations for building and querying an LTS; we will also need an operation for obtaining all the elements from an LTS and one for creating an empty LTS. We begin by stating precisely what functionality we expect of the LTS solution of the static Nearest Neighbor problem:

```
aux var St: MultiSet(point);
initially { St = { } };

func QueryS(x:point, S:LTS):distance
  post { result = dist(x,S.St) };

func Build(L:List(point)):LTS
  post {x∈result.St ≡ x∈L ∧ card(result.St)=length(L)};

func UnBuild(S:LTS):List(point)
  post {x∈result ≡ x∈S.St ∧ length(result)=card(S.St)};

func Empty: LTS
  post { result.St = { } };
```

In other words, an LTS acts as if it were a multiset of points. (Think of "aux var" as meaning "is modelled as".) Initially (immediately after declaration) the LTS is empty. Four routines are available; their effects are explicated in **post** conditions, or predicates that describe properties of the results in terms of properties of the inputs. In these predicates, qualified names are used to associate the modelling multiset with a *particular* LTS; thus "S.St" refers to the multiset that models the LTS named S and (in Build) "result.St" is the multiset that models the output of the function. *QueryS* is passed a point x and an LTS S and returns the distance from x to its nearest neighbor in S. *Build* is passed a collection (implemented as a list) of points and builds them into an LTS. *Unbuild* performs the inverse operation of build--extracting the points from the LTS and returning them in a list. *Empty* produces an empty LTS.

We can now show how the static Lipton-Tarjan structure can be transformed to yield a new dynamic structure for nearest neighbor searching (the transformation is due to Bentley and Saxe [1978]). We call this structure the *DNN*, for *Dynamic Nearest Neighbor*. Our objective is to provide definitions for operations with the following properties:

```
aux var D: MultiSet(point);
initially { D = { } };

func Init:DNN
  post { result.D = { } };

func QueryD(x:point, F:DNN):distance
  post { result = dist(x,F.D) };

proc Insert(var F:DNN, f:point)
  post { F.D = F'.D ∪ {f} };
```

Again, the data structure is modelled as a multiset and the effects of the functions are asserted as **post** conditions.

[1] Whenever we refer to the nearest neighbor problem we actually mean the related problem asking for the distance to the nearest neighbor. A simple bookkeeping operation allows the point realizing that minimum distance to be returned also.

Note that the value of F produced by Insert depends on the input values and that the input value is "primed" to distinguish it from the output value. Thus the post condition of Insert might be read, "the final value of F may be modeled as a multiset whose value is the union of the multiset that models the input value of F with the singleton set containing f."

3.2. Implementation

In the previous section we specified precisely the properties we assume of the Lipton-Tarjan Structure and those we must provide in the DNN structure. In this section we will describe an implementation for achieving these properties. We will accomplish this by first describing the implementation informally in words, and then specifying it more precisely in Alphard.

Our dynamic structure will consist of an array of static LTSs, each of a distinct size which is a power of two. Our structure is initially empty. When the first point is inserted, we build an LTS of size one. When the second point is inserted, we build the two points into an LTS of size two, discarding our previous LTS of size one. When the third point is inserted we have two LTSs (of sizes one and two), and after the fourth point is inserted we have one LTS of size four. Insertions proceed in a way analogous to binary counting: after the n-th element is inserted the structure includes an LTS of size 2^j if and only if the j-th bit of the binary integer n is one. Figure 1 illustrates the binary nature of our dynamic nearest neighbor structure. To answer a nearest neighbor query for a new point x we search for the nearest neighbor to x in each of the LTSs in our structure, and then return the point with the minimum distance of all those as the answer to the query. As an example, consider the case in which 79 elements are stored in the dynamic structure; we will then have static structures of sizes 64, 8, 4, 2, and 1. To find the nearest neighbor among the current set of points to point x we find x's nearest neighbors in each of the five structures and then return that point of the five realizing the minimum distance. To insert the new (80-th) point, we "take apart" the structures of size 8, 4, 2, and 1, and combine the points in those structures together with the new point into a new LTS of size 16; the 80 points are now stored in structures of size 64 and 16.

Assuming access to the four static subroutines mentioned above we will now formally describe three new subroutines that together define our new dynamic structure DNN. In

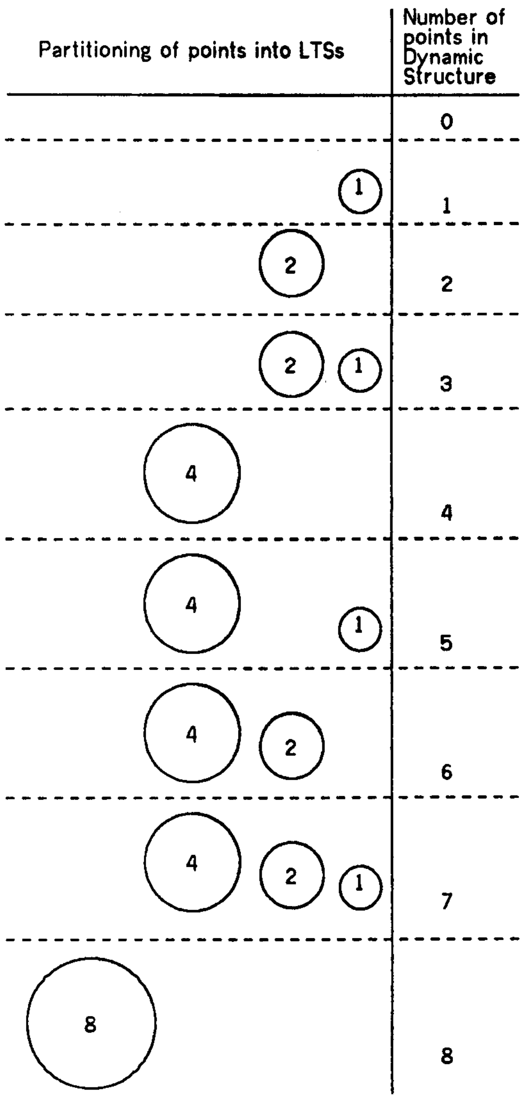
Partitioning of points into LTSs
Number of points in Dynamic Structure
0
1
1
2
2
2
1
3
4
4
4
1
5
4
2
6
4
2
1
7
8
8

Figure 1. History of dynamic structure: 0 through 8 points.

defining these routines we will make use of a one-way-infinite array P satisfying the property that for non-negative k the element P[k] is either an empty LTS or one containing 2^k points. In the program we will declare an integer and an unbounded vector (a "flexible vector") of LTSs with initial index 0 as follows:

var P: FlexVec (LTS, 0), High: integer;

In order to establish the correspondence of the assertions which model DNNs as multisets to this data structure and the programs that manipulate it, we must provide a function that shows how to interpret any instance of the data structure as a multiset. This is traditionally (if inaccurately) called a representation mapping. For the DNN structure it is

repmap $\{D = \bigcup_{i=0}^{High-1} P[i]\}$.

At any point only some fixed portion of the array P will point to LTSs; the value of the integer High is always one greater than the largest i such that P[i] points to a non-empty structure. For ease of implementation we maintain the invariant that $High \geq 0$ and require that P[High] always be an empty structure. As noted above, we require each element either to be empty or to contain a suitable number (2^k) of points. We can formally describe this invariant of our structure by the expression

$$(High \geq 0) \wedge (P[High] = \{\ \}) \wedge$$
$$(0 \leq k < High \supset (P[k] = \{\ \} \vee card(P[k]) = 2^k))$$

in which each term corresponds to one of the above conditions.

With this background we can now define the new subroutines. *Init* initializes the structure to contain no points:

```
proc Init(F:DNN) is
  begin F.High := 0; F.P[0] := Empty end
```

QueryD is the dynamic query subroutine; it accumulates the results of the static query (obtained by calling QueryS) on the individual static structures, keeping track of the minimum distance encountered. The syntax "**for** i **from** upto (1,F.High) **do**" expresses the counting loop traditionally written "**for** i:= 1 **step** 1 **until** F.High **do**". The routine is:

```
func QueryD(x: point, F:DNN): distance is
  value A:distance of
  A := QueryS(x,F.P[0]);
  for i from upto(1,F.High-1) do
      A := min(A,QueryS(x,F.P[i])
      od
  fo
```

Insert adds a new point to the DNN. It finds the lowest-numbered empty structure and replaces it with a structure composed of the elements in the lower-numbered structures and the new point:

```
proc Insert(f: point, F:DNN) is
  begin
  S := Listify(f);
  i := 0;
  while F.P[i] ≠ Empty do
      S := Concat(S, Unbuild(F.P[i]));
      P[i] := Empty;
      i := i + 1
      od;
  F.P[i] := Build(S);
  if i = F.High then
      F.High := F.High + 1; F.P[F.High] := Empty fi
  end
```

The functions Listify and Concat have the obvious meanings; they are formally specified in Section 4.4.

3.3. Correctness and Performance

The correctness of our dynamic nearest neighbor structure DNN can be established if we assume the correctness of Lipton and Tarjan's static structure. The insertion algorithm of our structure ensures that the set of points currently stored in the structure will be partitioned into sets which are then stored as LTSs; at any moment every point stored in the dynamic structure is stored in one and only one of the static structures. We can use this fact together with the correctness of the LTS and the fact that min correctly "combines" nearest neighbors to prove the correctness of the QueryD routine. The formal proof of this can be achieved as a specialization of the proof that will be given in Section 4.

To analyze the efficiency of our dynamic structure DNN we note that if the structure contains n elements then we are using at most $1 + \lg n$ LTSs. Since each is of size less than n we know that we can perform a nearest neighbor search on any of them in $O(\lg n)$ time. The total time required to search all $1 + \lg n$ LTSs is therefore $Q(N) = O(\lg^2 n)$. Since each of the LTSs requires space linear in the number of elements it contains, the total space requirement of the DNN is also linear. To count the total cost of having inserted n elements into a DNN is a bit more difficult. In the process of building a DNN of n elements (where n is one less than a power of two), a structure of size $m=2^j$ is built $(n+1)/2^{j+1}$ times. (This is the number of times the j-th bit in a binary word turns from zero to one in counting from zero to n.) For n one less than a power of two a simple sum shows that $P(n) = O(n \lg^2 n)$. We can use this fact to show that P(n) is of the same order when n is not one less than a power of two.

3.4. Summary of Dynamic Nearest Neighbor Searching

In the next section we will see how the transformation that allowed us to convert the nearest neighbor structure can be applied to a number of other data structures. Before we proceed to the general case, however, we will review the important steps in the conversion of the nearest neighbor searching structure from static to dynamic. On the most basic level, we were (presumably) given *code* that implemented the Lipton-Tarjan structure, and we then produced new code for the dynamic structure. The *correctness* proof for this structure can be obtained if we assume the correctness of the underlying LTS; we sketched this above and we will prove this formally in Section 4.5. Finally, we were given the *complexity* analysis of Lipton and Tarjan's structure, that

$P(n) = O(n \lg n)$, and
$Q(n) = O(\lg n)$

and we showed that our new structure has performance

$P(n) = O(n \lg^2 n)$, and
$Q(N) = O(\lg^2 n)$

(where P(n) now denotes the total time required to insert n elements). Notice that both the query times and the processing times of our new structure have increased by a factor of O(lg n); careful analysis (see Bentley and Saxe [1978]) shows that both factors are equal to (lg n)/2, plus lower order terms. (To emphasize the efficiency of our new structure compared to the old, consider the case n = 10,000--then (lg n)/2 is approximately seven.)

4. The General Transformation

In Section 3 we saw how a static data structure for nearest neighbor searching could be *transformed* into a dynamic structure for the same problem. In this section we will see how the same transformation can be used to convert a number of static search structures into dynamic ones. This transformation applies to a class of problems called the *decomposable searching problems*; we define this class in Section 4.1. The formal specifications of data structures for the static and dynamic cases are then given in Sections 4.2 and 4.3. An Alphard form implementing the transformation is given in Section 4.4, and is proved and analyzed for efficiency in Sections 4.5 and 4.6.

4.1. Decomposable Searching Problems

The transform that allowed us to convert a static nearest neighbor searching structure into a dynamic one can be used to convert a number of structures from static to dynamic. The essential point in the structure we saw above is that the nearest neighbor *problem* (not the Lipton-Tarjan *structure*) has the following property: we could partition the set F into subsets, answer the query on the subsets, and then combine those answers to yield an answer to the original problem. We call this property *decomposability*. We will identify a searching problem P by the kind of query it asks--we write Query(x,F) for the query asking the relation of object x to set F. Formally, a searching problem P is said to be decomposable if its query satisfies the condition

$\text{Query}(x, A \cup B) = \text{box}(\text{Query}(x,A), \text{Query}(x,B))$

for some binary operator, "box", that is computable in constant time. (We use the name "box" for the operator both for consistency with the original presentation in Bentley and Saxe [1978] and to avoid any assumption that it is a particular operator.)

All of the searching problems that we have mentioned so far are decomposable. Nearest neighbor searching is decomposable because nearest neighbor queries satisfy the property

$\text{NN}(x, A \cup B) = \min(\text{NN}(x,A), \text{NN}(x,B))$.

It was this decomposability of nearest neighbor that allowed us to split the points in our dynamic nearest neighbor structure DNN into separate LTSs and then combine answers on those to form an answer to the original problem. Likewise, Member searching is decomposable because it satisfies

$\text{Member}(x, A \cup B) = \text{or}(\text{Member}(x,A), \text{Member}(x,B))$.

The problem of Range searching is also decomposable, satisfying

$\text{Range}(x, A \cup B) = \cup(\text{Range}(x,A), \text{Range}(x,B))$.

(Here x is a rectangle and a range search returns all points in the planar point set that lie within that rectangle.) Bentley and Saxe [1978] have identified over twenty other searching problems that are decomposable.

4.2. Specifications of the Static Problem

The properties of the dynamic searching structure as defined in Alphard depend critically on the properties of the underlying static structure. In this section we state the properties required of the static structures somewhat more explicitly than we did in Section 4.1. This will motivate the precise specifications of the static problem that will be given in the next section.

The discussion above dealt informally with the types of the various components of a searching problem. A correct program must be much more precise. Three data types are involved:

- T1: the "query objects", or the subjects of searches.
- T2: the "stored objects", or the elements of the set in which the search is carried out.
- T3: the answer to the query.

Three decomposable searching problems were discussed in Section 4.1. The types of the components and the operations box and Q are summarized in the following table.

Problem	Query Objects (T1)	Stored Objects (T2)	Query Answer (T3)	Box	Q(x,F)
Member	real	real	boolean	$\vee$	$\bigvee_{f \in F} =(x,f)$
Distance to Nearest Neighbor	point	point	distance	min	$\mathrm{MIN}_{f \in F}\, \mathrm{dist}(x,f)$
Range	rectangle	point	point set	$\cup$	$\bigcup_{f \in F} \mathrm{encloses}(x,f)$

A complete definition of a static problem must include programmed routines for searching a static structure, constructing a static structure, extracting the stored objects from a static structure, creating an empty static structure, and determining whether one is in fact empty. In addition, the problem definition must supply a function for computing the box operation and a guarantee that the query and box operations jointly satisfy the assumption of decomposability. With the above discussion of the required types, we can specify what we assume *in general* about a static solution:

```
form Stat(T2:form) is
  specs
  aux var St: Multiset(T2);
  initially { St = { } };
  func QueryS(x:T1, S:Stat(T2)):T3
      post { result = Q(x,S.St) };
  func Build(L:List(T2)):Stat(T2)
      post { x∈result.St ≡ x∈L ∧ card(result.St)=length(L) };
  func UnBuild(S:Stat(T2)):List(T2)
      post { x∈result ≡ x∈S.St ∧ length(result)=card(S.St) };
  func Empty: Stat(T2)
      post { result.St = { } };
  func IsEmpty(S:Stat(T2)):boolean
      post { result ≡ S.St = { } }
  end;

func box(a1,a2:T3):T3;

aux func Q(x:T1,r:Set(T2)):T3
axiom { [r = p ∪ q] ⊃ [Q(x,r) = box(Q(x,p), Q(x,q))] };
```

4.3. Specifications of the Dynamic Problem

We now turn to the definition of dynamic structure Dynamic. It needs these properties:

```
aux var D: Set(T2);
initially { D = { } };

func QueryD(x:T1, F:Dynamic):T3
  post { result = Q(x,F.D) };

proc Insert(var F:Dynamic, f:T2)
  post { F.D = F'.D ∪ {f} };
```

The general transform for converting a static structure for a decomposable searching problem into a dynamic structure can now be described. Bear in mind that the dynamic nearest neighbor structure DNN was produced by applying this general transform to the particular case of the Lipton-Tarjan (static) nearest neighbor searching structure.

At any given point a Dynamic will consist of a set of Stats whose sizes are distinct powers of two. Inserting a new element into a Dynamic is analogous to incrementing a binary integer--starting with the structure of size one, we look at successively larger structures until one is missing, at which point we use BuildS to make a new structure whose size is that power of two. We can then query the Dynamic by searching each Stat with QueryS; because the problem is decomposable the given box operator can be used to combine these answers. In the next section we will make this informal description precise.

4.4. Alphard Program for Dynamic Structures

We now address the exact implementation of the transform, casting it as an Alphard **form**. We gather the specifications given above and combine them with the code for the implementation. Although the specifications for this program deal only with the functionality, or input-output relations of the operations involved, we are also able to enforce the properties of the data structure that are needed to achieve the efficiency we require.

From the description of Alphard given by Wulf, London, and Shaw [1976] we know that a **form** has two major components: a specification (for public consumption) and an implementation (of interest only to the implementor). In addition, a parameterized **form** definition must state explicitly what is assumed about the parameters. The power of Alphard to deal with the very general definition developed here stems from the fact that the parameter to a **form** may be another definition (i.e., another **form**, not simply a value).

```
form Dynamic(Static:form)
    assumes form Static(T1,T2,T3: form) is
        specs
        form Stat(T2:form) is
            specs
            aux var St: MultiSet(T2);
            initially { St = { } };
            func QueryS(x:T1, S:Stat(T2)):T3     post { result = Q(x,S.St) };
            func Build(L:List(T2)):Stat(T2)      post { $x \in$ result.St $\equiv$ $x \in L$ $\wedge$ card(result.St)=length(L) };
            func UnBuild(S:Stat(T2)):List(T2)    post { $x \in$ result $\equiv$ $x \in$ S.St $\wedge$ length(result)=card(S.St) };
            func Empty: Stat(T2)                 post { result.St = { } };
            func IsEmpty(S:Stat(T2)):boolean     post { result $\equiv$ S.St = { } }
            end;

        func box(a1,a2:T3):T3;
        aux func Q(x:T1,r:MultiSet(T2)):T3
        axiom { $r = p \cup q \supset$ Q(x,r) = box(Q(x,p), Q(x,q)) };
        end

is specs
aux var D: MultiSet(T2);
initially { D = { } };
func QueryD(x:T1, F:Dynamic):T3              post { result = Q(x,F.D) }
proc Insert(var F:Dynamic, f:T2)             post { F.D = F'.D $\cup$ {f} }

impl
var P: FlexVec(Stat(T2),0), High: integer  init begin High := 0; P[0] := Empty end;

repmap { D = $\bigcup_{i=0}^{High-1}$ P[i] };

invariant { High$\geq$0 $\wedge$ P[High] = { } $\wedge$ $0 \leq k <$ High $\supset$ (P[k] = { } $\vee$ card(P[k]) = $2^k$) };

func QueryD(x:T1, F:Dynamic):T3              post { result = Q(x, $\bigcup_{j=0}^{F.High-1}$ F.P[j]) };
    is value A: T3 of
    A := QueryS(x,F.P[0]);
    for i from upto(1,F.High-1)
        do  A := box(A, QueryS(x,F.P[i]));  assert { A = Q(x, $\bigcup_{j=0}^{i}$ F.P[j])  $\wedge$  F.P=F'.P $\wedge$ F.High=F'.High }  od
    fo;

proc Insert(var F:Dynamic, f:T2)             post { $\bigcup_{j=0}^{F.High-1}$ F.P[j] = $\bigcup_{j=0}^{F'.High-1}$ F'.P[j] $\cup$ {f} };
    is begin
    var S: List(T2), i: integer;
    S := Listify(f);  i := 0;
    while $\neg$ IsEmpty(P[i])

        assert { {f} $\cup$ $\bigcup_{j=0}^{F'.High-1}$ F'.P[j] = Setify(S) $\cup$ $\bigcup_{j=i}^{F.High-1}$ F.P[j]  $\wedge$  $\bigcup_{j=0}^{i-1}$ F.P[j] = { }
                $\wedge$ length(S) = $2^i$  $\wedge$  High$\geq$0 $\wedge$ P[High] = { } $\wedge$ $0 \leq k <$ High $\supset$ (P[k] = { } $\vee$ card(P[k]) = $2^k$) }

        do  S := Concat(S,UnBuild(F.P[i]));  F.P[i] := Empty;  i := i+1  od;
    F.P[i] := Build(S);
    if i = F.High then F.High := F.High + 1; F.P[F.High] := Empty fi
    end
end
```

Figure 2. The Alphard form Dynamic.

The Alphard **form** definition given below defines operations for dynamic search problems. In order to use this **form**, called "Dynamic", the definitions related to the static structure and to the crucial operations on the static structure must have been encapsulated in another **form**. The specifications for the latter **form** capture exactly the necessary properties of the static structure and its associated definitions; the minimum set of properties for the static problem is specified as an assumption in the definition of the dynamic structure.

The definition of Dynamic is therefore *generic*; that is, the definition of the dynamic structure Dynamic is parameterized by the definition of the static problem (with formal name Static). The precise assumptions that the static definition must satisfy in order for the **form** to be used (sketched in Section 4.2) are given in an "**assumes**" clause. Briefly, these are a suitable data structure for the static problem and an operator "box" for combining the results of the static queries; the query function for the data structure and the box function must jointly satisfy the decomposability property, specified here as an axiom.

The properties of both the static and dynamic structures are specified abstractly by modelling the structures as multisets. In both cases, the abstract specification (**spec** part) of the **form** contains routine headers and assertions couched in terms of multisets (and, in the case of Stat, sequences). The information provided here is sufficient to use the Dynamic definition.

An implementation is given only for the dynamic structure. This implementation is based on an extendible vector of static structures (a "FlexVec"); the concrete specifications reflect this. The implementation (**impl** part) of Dynamic gives both the Alphard code for implementing a dynamic structure and concrete specifications: assertions about the properties of the data structure that must be preserved (**invariant**), about the effects of the two routines (**post**), and about properties that must hold during execution of the routines (**assert** -- e.g., loop invariant). In the concrete specifications, the P[i] may be treated as multisets because Stats are abstractly specified in terms of multisets. Note also that when an assertion (**pre**, **invariant**) is **true**, it is omitted from the program. The complete Alphard **form** implementing Dynamic is shown in Figure 2.

The implementation depends on two other nonscalar types, List (linear lists) and FlexVec (vectors with flexible upper bounds). The portions of the specifications of these two types that we assume here are

```
form List(T:form) is
   specs
   aux var Ls: sequence(T);
   initially { Ls = <> };
   func Listify(x:T):List
        post { result.Ls = <x> };
   func Concat(a,b:List):List
        post { result.Ls = a.Ls ~ b.Ls };
        note property. of sequences:
            length(result) = length(a) + length(b) eton
   aux func length(a:List):integer
        post { result = length(a.Ls) };
   aux func &∈(x:T, L:List):boolean
        post { ∃i st x=L.Ls[i] };
   aux func Setify(L:List(T2)):Set(T2)
        post { x∈result ≡ x∈L.LS ∧ card(result)=length(L.Ls) };
   end

form FlexVec(T:form, lb:integer) is
   specs
   aux var Fl:sequence(T);
   selector &subscript(F:FlexVec, k:integer)
        pre { k ≥ lb }  post { result = F.Fl[k-lb+1] };
   end
```

Formal definitions of sequences and sets are assumed here and in the definition of Dynamic.

4.5. Correctness Proof

The correctness of the dynamic structure can be seen intuitively if we assume the correctness of the underlying static structure. The insertion algorithm guarantees that if the n query objects have been inserted into our dynamic structure, then the set represented by the structure has been partitioned into subsets whose sizes are distinct powers of two, and that each of these is represented by a Stat. The essential point here for the correctness of Dynamic is that we have a partitioning of the set of query objects (each point in S is stored in exactly one Stat)--the powers of two are important only in establishing the efficiency of the dynamic structure. We have thus sketched that the insertion algorithm maintains a "reasonable" representation of its set of inputs. Given this, it is easy to show that the search algorithm returns the correct value. We have assumed that the underlying Stat is in fact correct, so we know that each of the (up to) lg n searches correctly answers the query Q in each of the sets of the partitioning. The decomposability property stated in the **axiom** of the Static specification then guarantees that the correct answer for the complete set is computed.

The above sketch makes the validity of the transform believable; we will now show how given the formal correctness of Static we can formalize the arguments we have just sketched to show rigorously the correctness of our Dynamic structure. Several steps are required to verify an Alphard form. We must show that the representation is valid and that each instance is initialized properly; we will demonstrate these facts rigorously in Sections 4.5.1 and 4.5.2. In addition, for each operation (Insert and QueryD) we must both verify the concrete assertions about its implementation and show a proper relation between concrete and abstract specifications. In Section 4.5.3 we will sketch these proofs; the full details of the proofs can be found in Appendix I. All the proofs up to this point establish only weak correctness; we examine the issue of termination in Section 4.5.4.

Certain notations will be employed throughout this section. We use the following abbreviations for standard assertions in the programs:

β_{pre}	abstract precondition
β_{post}	abstract postcondition
β_{in}	concrete precondition
β_{out}	concrete postcondition
I_a	abstract invariant
I_c	concrete invariant
I_{loop}	loop invariant
β_{init}	initially clause from abstract specs

This usage is drawn from the description of Alphard methodology given by Wulf, London, and Shaw [1976]. The notation

$$P \{ S \} R$$

states the theorem, "for assertion R to hold after execution of statement S, it is sufficient for P to hold beforehand". We use predicate transformers in the proofs, so this is equivalent to

$$P \supset wp(S,R).$$

4.5.1. Validity of the Representation

We must first show that any configuration of the data (P and High) that can result from operations of this form (i.e., any configurations that satisfy the concrete invariant) will correctly represent a dynamic data structure. If x denotes any instance of a Dynamic, the theorem to prove is

$$I_c(x) \supset I_a(\mathbf{repmap}(x)).$$

The abstract invariant admits any set (i.e., $I_a \equiv \mathbf{true}$), so the theorem is vacuously true.

4.5.2. Initialization of an Object

We must next show that the object is initialized correctly. Let C_{init} denote the code for initialization and x a concrete object (i.e., an instantiation of Dynamic). The theorem to prove is then

$$\mathbf{true} \{ C_{init} \} \beta_{init}(\mathbf{repmap}(x)) \wedge I_c(x)$$

or

$$\mathbf{true} \{ \text{High} := 0;\ P[0] := \text{Empty} \}$$

$$\bigcup_{i=0}^{High-1} P[i] = \{ \} \wedge \text{High} \geq 0 \wedge P[\text{High}] = \{ \}$$

$$\wedge\ 0 \leq k < \text{High} \supset (P[k] = \{ \} \vee \text{card}(P[k]) = 2^k)$$

Computing weakest preconditions for the assignment statements yields

$$(\bigcup_{i=0}^{-1} P[i] \cup \{ \}) = \{ \} \wedge 0 \geq 0 \wedge \text{Empty} = \{ \}$$

$$\wedge\ 0 \leq k < 0 \supset (P[k] = \{ \} \vee \text{card}(P[k]) = 2^k)$$

or

$$\{ \} = \{ \} \wedge 0 \geq 0 \wedge \{ \} = \{ \}$$

$$\wedge\ (\mathbf{false} \supset (\{ \} = \{ \} \vee \text{card}(\{ \}) = 0))$$

This is clearly true.

4.5.3. Overview of the Verification of the Routines

Having established the validity of representation and the proper initialization of a Dynamic, our next step in the verification of the form is the correctness of the routines that operate on the form. For each routine we must show two things: the correctness of the concrete program and the correspondence of the concrete program to the abstract specification. In this section we will sketch such proofs for routines QueryD and Insert; as mentioned before, the full details of the proofs can be found in Appendix I.

In treating routine QueryD our first task is to prove the concrete correctness of the program. The theorem to prove is

$$\beta_{in}(x,F) \wedge I_c(F)\ \{ \text{body} \}\ \beta_{out}(x,F) \wedge I_c(F).$$

In words, we must show that if both the concrete precondition for routine QueryD and the concrete invariant for the Dynamic structure are known to hold before execution of QueryD, then afterwards both the concrete postcondition of the routine and the concrete invariant of the structure will hold. Showing that the invariant is maintained is trivial: the routine does not alter the structure. Showing that the concrete postcondition follows from the concrete precondition and structural invariant is a fairly straightforward exercise in program verification. The crucial observation in that proof is the place of the decomposability axiom. It allows us to assert that combining the current answer with the response to a new query via the box operator yields the correct answer to a query on a larger subset of the stored elements. This shows that the invariant holds through each iteration of the loop, and the rest of the proof is rather mechanical.

The second step in the verification of QueryD is to show the relation between the concrete and abstract assertions. The first theorem to be proved is

$$I_c(F) \wedge \beta_{pre}(x,\mathbf{repmap}(F)) \supset \beta_{in}(x,F)$$

This says that the concrete precondition of the structure and the abstract precondition of the routine must together imply the concrete precondition of the routine. For the case of QueryD this is trivial because the concrete precondition is **true**. The second representation theorem to show is

$$I_c(F) \wedge \beta_{pre}(x,\mathbf{repmap}(F')) \wedge \beta_{out}(x,F)$$
$$\supset \beta_{post}(x,\mathbf{repmap}(F)).$$

In words this says that the concrete invariant, the abstract precondition (predicated of the original structure), and the concrete postcondition together imply the abstract postcondition. The proof of this follows from the representation mapping.

The verification of routine Insert proceeds in the same two steps. The concrete correctness of the routine is again just a rather tedious exercise in verification techniques; the complicated part is showing that the invariant is preserved through the loop. The second step of the proof is the relation between abstract and concrete assertions; this step again reduces to observations about similarities in the representation mapping and the conditions.

4.5.4. Termination

Thus far in this correctness proof we have confronted only the question of weak correctness--that if the program terminates, then it will produce output in accordance with specifications. In this section we will sketch how all the routines of a Dynamic can formally be shown to terminate. The termination of the initialization is trivial since the procedure consists of only two statements (with no iterations). The termination of QueryD is almost as easy if we assume the termination of QueryS--the only iteration is in a **for** loop, which must always halt. To show the termination of Insert we must assume the termination of Build and UnBuild. We must then show that the **while** loop of routine Insert always halts. This is guaranteed by the invariant, however, which asserts that P[High] is always an empty Stat. The arguments that we have just sketched can be formalized to show rigorously the strong correctness of the Dynamic structure.

4.6. Performance

The termination of the routines of Dynamic assures us only that they will indeed halt after some finite number of steps; in this section we will count the time complexity of the routines. In the conversion from the static nearest neighbor structure to the dynamic (in Section 3.3) we saw that factors of lg n were added to both query and processing times. That particular example was indicative of the general case. Bentley and Saxe [1978] have shown that if $Q_S(n)$ is the time for searching a Stat containing n elements, then the time for searching a Dynam of n elements is bounded above by

$$Q_D(n) \le Q_S(n)\ (\lg n)$$

(as long as Q_S is monotone increasing). Bentley and Saxe have also shown that if $P_S(n)$ (the time to build a static structure of n elements) grows at least linearly, then the time for inserting the first n elements into a Dynam will be bounded above by

$$P_D(n) \le P_S(n)\ (\lg n).$$

These bounds are somewhat pessimistic; a more careful analysis shows that the factors of lg n can be considerably reduced in many problems.

5. Conclusions

We have described and specified a general ***generic*** definition that is applicable in a specific (but broad) problem domain. The definition, in effect, takes the solution to one problem as a parameter and automatically yields a solution to a related problem. Specifically, we mapped solutions of static searching problems to solutions of dynamic ones.

This work also shows the efficacy of formal specification and verification techniques for problems in algorithm design. The formal specification makes precise the domain in which the transformation is applicable. The fact that the **form** is verified allows the transformation to be applied with confidence -- and without knowledge of the implementation of either the static solution or of the transformation itself. These advantages were emphasized during the development of the form given here: the abstract algorithm was reformulated in several important ways while the formal definition was being developed.

The static-to-dynamic transformation illustrates a general paradigm. Bentley and Saxe [1978] have developed a spectrum of similar transformations that allow tradeoffs to be made in the factors added to processing and query times. Such a spectrum is very useful if we know *a priori* the expected ratio of queries to insertions in the particular application for which we are designing the structure. Bentley and Saxe have also described two other completely different transformations for decomposable searching problems. One of these transforms allows queries to be further specified by "range variables" and the other transform can be used to make tradeoffs between processing and query times in a very general framework. Each of these three transforms have been used to yield new results in concrete complexity. Current plans call for formalizing all of these additional transforms as Alphard forms and rigorously verifying them. This will lead to a set of software engineering tools as well as a set of components useful to the algorithm designer in building correct and efficient algorithms with known complexity.

This paper has demonstrated a technique for writing parameterized abstractions that are much more general than procedures. The ability to specify formally the minimal assumptions made about the parameter was crucial to the development. This approach has the potential to affect the practice of software engineering, both by supporting a library or handbook of software tools with known properties and by making the algorithm descriptions of concrete complexity more precise. Naturally, much research remains to be done before achieving those goals. The present work does, however, show that ideas and results can cross the boundary between complexity and software engineering.

Acknowledgements

We would like to express our appreciation to Ron Brender, Dorothea Haken, Mike Horowitz, Jim Saxe, and Bill Wulf for helpful conversations and comments on early drafts and proofs.

References

Bentley, J. L. and J. B. Saxe [1978]. Decomposable searching problems, in preparation. Summary available in "Decomposable searching problems," by J. L. Bentley, Carnegie-Mellon Computer Science Report CMU-CS-78-145.

Hilfinger, P. N. *et al* [1978]. "(Preliminary) An Informal Definition of Alphard", Carnegie-Mellon Computer Science Report CMU-CS-78-105.

Knuth, D. E. [1973]. ***The Art of Computer Programming, volume 3: Sorting and Searching.*** Addison-Wesley, Reading, Mass.

Lipton, R. J. and R. E. Tarjan [1977]. "Applications of a planar separator theorem," ***Eighteenth Symposium on the Foundations of Computer Science***, pp. 162-170, IEEE.

Wulf, W. A., R. L. London, and M. Shaw [1976]. "An introduction to the construction and verfication of Alphard programs," ***IEEE Transactions on Software Engineering*** *SE-2*, 4, December 1976, 253-265.

I. A Detailed Correctness Proof

In this appendix we will provide the details of the proofs we sketched in Section 4.5. We will employ here the same notations as in that section. In the first part of this appendix, Section I.1, we will prove the correctness of the QueryD routine. We then prove the correctness of the Insert routine in Section I.2.

I.1. Verification of Routine QueryD

We perform the proof of this routine in two steps, one for the correctness of the concrete program and one for its correspondence to the abstract specification. The preconditions for both routines are **true**, the abstract postcondition is expressed in terms of sets, and the concrete postcondition is expressed in terms of the FlexVec P and the integer High.

Concrete correctness: The theorem to prove is

$\beta_{in}(x,F) \wedge I_c(F)$ { body } $\beta_{out}(x,F) \wedge I_c(F)$

Note that the routine does not modify F.P or F.High. As a result, I_c must remain true (the loop invariant contains terms to assure that this can be done formally). We will consider here only the verification of β_{out}. The proof follows the same general plan as will the proof for Insert. We begin with the loop invariant, and then show that the initialization statement leads correctly to the loop.

Let Z denote the loop body

A := box(A, QueryS(x,F.P[i]));

and let $I_{loop}[1..i]$ denote the loop invariant

$A = Q(x, \cup_{j=0}^{i} F.P[j]) \wedge F.P=F'.P \wedge F.High=F'.High$

To show the preservation of the loop invariant, we must show

$I_{loop}[1..i-1] \supset wp(Z, I_{loop}[1..i]) \wedge I_{loop}[1..F.High-1] \supset \beta_{out}$

The loop does not modify F.P or F.High, so we ignore the identity terms. For the first of the remaining terms we compute the weakest precondition of the loop body by substituting for the assignment statement of Z, yielding

$box(A, QueryS(x,F.P[i])) = Q(x, \cup_{j=0}^{i} F.P[j])$

Separating the last term of the union on the right-hand side from the remainder of the union and applying the specification axiom of Q, we reduce this to

$box(A, QueryS(x,F.P[i])) = box(Q(x, \cup_{j=0}^{i-1} F.P[j]), Q(x, F.P[i]))$

Applying the postcondition of QueryS, we obtain

$box(A, Q(x,F.P[i])) = box(Q(x, \cup_{j=0}^{i-1} F.P[j]), Q(x, F.P[i]))$

which is true provided

$A = Q(x, \cup_{j=0}^{i-1} F.P[j])$.

But this is precisely $I_{loop}[1..i-1]$, so the first term of the condition is demonstrated. Next, note that the exit condition for the loop, $I_{loop}[1..F.High-1]$, is

$A = Q(x, \cup_{j=0}^{F.High-1} F.P[j])$

The **value** expression associates A with the function result, so the second term of the condition is demonstrated.

To complete the proof of the routine it remains to show that

$\beta_{in}(x,F) \wedge I_c(F)$ { A := QueryS(x,F.P[0]) } $I_{loop}[1..0]$

which follows by direct substitution and appeal to the specification of QueryS.

This proves that the body of QueryD implements its concrete assertions.

Relation between concrete and abstract assertions: Two theorems must be proved:

$$I_c(F) \wedge \beta_{pre}(x,\text{repmap}(F)) \supset \beta_{in}(x,F)$$

$$I_c(F) \wedge \beta_{pre}(x,\text{repmap}(F')) \wedge \beta_{out}(x,F) \supset \beta_{post}(x,\text{repmap}(F))$$

Since β_{in} is **true**, the first is vacuously true. Since β_{pre} is **true**, the second reduces to

$$[I_c(F) \wedge \textbf{result} = Q(x, \cup_{j=0}^{F.High-1} F.P[j])] \supset \textbf{result} = Q(x, \cup_{j=0}^{F.High-1} F.P[j])$$

That is, the abstract postcondition is a direct mapping of the concrete postcondition, and the theorem is proved.

I.2. Verification of Routine Insert

As in our proof of QueryD, we first show the correctness of the concrete program, and then the relation between the concrete and abstract assertions. The program has (implicitly) abstract and concrete preconditions of **true**. The abstract postcondition is given in the specification part of the form, near the procedure header; it is stated in terms of the set represented by the parameter F. The concrete postcondition is given in the implementation part of the form, just before the body; it is stated in terms of the P and High components of the parameter F and also in terms of the parameter f.

Concrete correctness: The theorem to prove is

$$\beta_{in}(F,f) \wedge I_c(F) \ \{ \text{ body } \} \ \beta_{out}(F,f) \wedge I_c(F)$$

or that the body satisfies its concrete specifications and preserves the concrete invariant.

We first show the correctness of the loop invariant, then work the postcondition backward through the program. We argue correctness by showing the state of the computation at three points: just after the loop, just before the loop, and at the beginning of the routine body. Let Z denote the loop body

```
S := concat(S,Unbuild(F.P[i]));
F.P[i] := Empty;
i := i+1;
```

and let I_{loop} denote the loop invariant

$$\{f\} \cup \cup_{j=0}^{F'.High-1} F'.P[j] = \text{Setify}(S) \cup \cup_{j=i}^{F.High-1} F.P[j] \wedge \cup_{j=0}^{i-1} F.P[j] = \{ \}$$

$$\wedge \ \text{length}(S) = 2^i$$

$$\wedge \ [\text{High} \geq 0 \wedge P[\text{High}] = \{ \}]$$

$$\wedge \ 0 \leq k < \text{High} \supset (P[k] = \{ \} \vee \text{card}(P[k]) = 2^k)$$

To show the correctness of the loop invariant, we must show

$$[\ \neg\ \text{IsEmpty}(F.P[i]) \wedge I_{loop}(i) \supset wp(Z, I_{loop}(i+1))\ ] \ \wedge \ [\ \text{IsEmpty}(F.P[i]) \wedge I_{loop}(i) \supset R\ ]$$

where R is the weakest precondition for the remainder of the routine (determined from the postcondition $\beta_{out} \wedge I_c$). The second term of this predicate captures the state of the computation after the loop; the first term establishes the state before the loop.

State of computation after loop: To show the second term of the theorem about the loop invariant, we need to determine R (the weakest precondition for the portion of the routine following the loop) from

```
R {  F.P[i] := Build(S);
     if i = F.High then F.High := F.High + 1; F.P[F.High] := Empty fi  }  β_out(F,f) ∧ I_c(F)
```

Computing weakest preconditions, we find R is

$[\cup_{j=0}^{F.High-1} F.P[j] - F.P[i] \cup Build(S) = \cup_{j=0}^{F'.High-1} F'.P[j] \cup \{f\}]$

$\wedge$ [Build(S) = { } $\vee$ card(Build(S))=2^i]

$\wedge$ [F.High=i $\wedge$ F.High+1$\geq$0 $\wedge$ Empty={ } $\vee$ F.High$\neq$i $\wedge$ F.High$\geq$0 $\wedge$ F.P[F.High]={ }]

$\wedge$ [0$\leq$k<F.High $\supset$ (F.P[k]={ } $\vee$ card(F.P[k])=2^k)]

Now to show the second term of the theorem, we must prove

F.P[i]={ } $\wedge$ I_{loop} $\supset$ R.

The terms of R correspond in order to the terms in I_{loop}. The first term follows because of the correspondence in the postconditions of Build and Setify. The second term follows because length(S) = card(Build(S)). The third and fourth terms follow directly.

State of computation before loop: The loop body consists entirely of assignment statements, so the weakest precondition wp(Z, I_{loop}) is found by back-substitution. After some manipulation of set expressions, it is

$\{f\} \cup \cup_{j=0}^{F'.High-1} F'.P[j]$ = Setify(Concat(S,UnBuild(F.P[i]))) $\cup$ $\cup_{j=i+1}^{F.High-1} F.P[j]$

$\wedge$ $\cup_{j=0}^{i} F.P[j]$ = { }

$\wedge$ length(Concat(S,UnBuild(F.P[i]))) = 2^{i+1}

$\wedge$ F.High$\geq$0 $\wedge$ P[F.High] = { } $\wedge$ 0$\leq$k<F.High $\supset$ (P[k] = { } $\vee$ card(P[k]) = 2^k).

(Recall that the primed identifiers refer to values at input to the routine.) From the specifications of Concat, Setify, and UnBuild, we know

Setify(Concat(S,UnBuild(F.P[i]))) = Setify(S) $\cup$ F.P[i]
length(Concat(S,UnBuild(F.P[i]))) = length(S) + card(F.P[i]).

Using these facts, wp(Z, I_{loop}) becomes

$\{f\} \cup \cup_{j=0}^{F'.High-1} F'.P[j]$ = Setify(S) $\cup$ F.P[i] $\cup$ ($\cup_{j=i+1}^{F.High-1}$ F.P[j] - F.P[i] $\cup$ { })

$\wedge$ ($\cup_{j=0}^{i}$ F.P[j] - F.P[i] $\cup$ Empty) = { }

$\wedge$ length(S) + card(F.P[i]) = 2^{i+1} $\wedge$ F.P[i] = Empty

$\wedge$ F.High $\geq$ 0 $\wedge$ F.P[F.High] = { } $\wedge$ (0$\leq$k<F.High $\wedge$ K$\neq$i) $\supset$ (F.P[F.High] = { }).

which simplifies to

$\{f\} \cup \cup_{j=0}^{F'.High-1} F'.P[j]$ = Setify(S) $\cup$ $\cup_{j=i+1}^{F.High-1}$ F.P[j] $\wedge$ ($\cup_{j=0}^{i}$ F.P[j] = { })

$\wedge$ length(S) = 2^{i+1}

$\wedge$ High$\geq$0 $\wedge$ P[High] = { } $\wedge$ 0$\leq$k<High $\supset$ (P[k] = { } $\vee$ card(P[k]) = 2^k)

The first term of the theorem about the loop invariant,

$\neg$ IsEmpty(F.P[i]) $\wedge$ I_{loop}(i) $\supset$ wp(Z,I_{loop}(i+1))

follows directly.

This establishes I_{loop} as a valid loop invariant.

State of computation at beginning of body: It remains to demonstrate that

β_{in}(F,f) $\wedge$ I_c(F) { S := Listify(f); i := 0 } I_{loop}.

Computing weakest preconditions, observing that Setify(Listify(f)) = {f}, and noting that length(<f>)=1, this becomes

$$\textbf{true} \wedge I_c(F) \supset [\ \{f\} \cup \bigcup_{j=0}^{F'.High-1} F'.P[j] = \{f\} \cup \bigcup_{j=0}^{F.High-1} F.P[j] \wedge 1 = 2^0\ ].$$

We have arrived at the beginning of the routine, so F'.High=F.High and F'.P=F.P; thus the right side of the implication is a tautology and the theorem is proved.

These three steps prove that the body of Insert implements its concrete assertions.

Relation between concrete and abstract assertions: Two theorems must be proved:

$$I_c(F) \wedge \beta_{pre}(\text{repmap}(F),f) \supset \beta_{in}(F,f)$$

$$I_c(F') \wedge \beta_{pre}(\text{repmap}(F'),f) \wedge \beta_{out}(F,f) \supset \beta_{post}(\text{repmap}(F),f)$$

Since β_{in} is true, the first is vacuously true. Since β_{pre} is true, the second reduces to

$$[I_c(F') \wedge \bigcup_{j=0}^{F.High-1} F.P[j] = \bigcup_{j=0}^{F'.High-1} F'.P[j] \cup \{f\}] \supset$$

$$[\bigcup_{j=0}^{F.High-1} F.P[j] = \bigcup_{j=0}^{F'.High-1} F'.P[j] \cup \{f\}]$$

That is, the abstract postcondition is a direct mapping of the concrete postcondition, and the theorem is proved.