Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars

Roger Hoover
Tim Teitelbaum

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

Aggregate valued attributes, which store collections of keyed elements, are required in attribute grammars to communicate information from multiple definition sites to multiple use locations. For syntax directed editors and incremental compilers, symbol tables are represented as aggregate values. We present efficient algorithms for incrementally maintaining these aggregate values and give an incremental evaluation algorithm that restricts attribute propagation to attributes dependent only upon information within the aggregate value that has changed.

1.0 Introduction

We are concerned with the problem of maintaining a consistent database of facts derived from a dynamically changing, hierarchically structured object. The object under consideration may be as small as the abstract syntax tree of an individual procedure, or as large as the directory structure of an entire file system.

As in [DRT81], we shall assume that the inferred database is represented as a collection of attributes decorating the nodes of the tree structured object, and that these attributes are the solution of a set of semantic equations given in an attribute grammar. An incremental update algorithm is used to reach a new solution after this attributed tree is modified by subtree replacement. Such a declarative specification of derived information offers the advantage that the sequence of steps needed to update the old database in response to each mutation of the object can be inferred from the dependency structure of the specification.

Because each nonterminal of an attribute grammar is associated with only a constant number of attributes, aggregate valued attributes must be used whenever information is to be communicated to and/or from an unbounded number of locations. The attributes that we are communicating from, called *definitions*, add elements to the aggregate value. Elements of the aggregate value are then selected by attributes called *uses*. A *key* is associated with each element of the aggregate value, and is used to match *definitions* with *uses*. This key is part of the definition of the element and is employed by each use to identify the element.

Semantic equations called *copy rules* are used to specify the equality between the aggregate value and *copy attributes*. This allows the aggregate value to be broadcast from the definitions throughout the portion of the derivation tree where uses may be located.

Symbol tables in the attribute grammar specifications of block structured languages are typically represented this way. A set of identifier definitions is collected and copied throughout the attributed derivation tree so that each identifier

use can be looked up in the set and value or type information may be obtained.

When a change is made to an aggregate valued attribute in the attributed tree, a propagation algorithm is used to insure that all attributes that are functionally dependent upon the aggregate value have values consistent with the new aggregate value. This set of dependent attributes includes all uses and is typically large, much larger than the set of attributes that depend only upon the changed information in the aggregate value.

The incremental evaluator of [R84] updates an attributed tree in O(|*AFFECTED*|) steps where *AFFECTED* is the set of attributes that change value. This time bound, however, is only achieved for aggregate values because the copy attributes that broadcast the aggregate value change as a result of the change in the aggregate value and are therefore in *AFFECTED*. The number of such copy attributes is proportional to the number of uses of the aggregate value.

In [H86], we showed how copy rules can be bypassed and how changes can be propagated directly from definitions to uses. Each use, however, is still functionally dependent upon the collection of definitions as a whole. Therefore, any change in the aggregate value will still propagate to all uses of that aggregate value regardless of whether or not the element used has been changed. This results in O(|*uses*|) steps to update the attributed tree after any definition change.

We solve this problem for keyed aggregate values by introducing a finite function type into our attribute grammar specification language. A finite function is a mapping in which all but a finite number of domain elements map to the same range element. After a change in the attributed tree, we are able to compute and propagate the set of bindings that are different in a given finite function. We construct data structures at attribute instances where finite function values are created that allow us to locate and propagate to only these uses of the finite function corresponding to bindings that have changed.

Using finite functions, the specification remains a totally declarative attribute grammar. Both synthesized and inherited attributes may have finite function values. The result is an efficient method of updating aggregate valued attributes, especially after small changes in the aggregate value.

The finite function manipulation algorithms described in this paper have been implemented in the Synthesizer Generator [RT84] . We have modified our syntax directed editor for Pascal to use finite functions to represent the symbol table. When a single global declaration is changed in a sample 500 line Pascal program, these modifications have resulted in a five fold reduction in the time required to update the attributed tree. This factor increases with larger programs.

2.0 Overview

In Section 3 of this paper, we review previous approaches to the aggregate problem. In Section 4, we introduce the finite function type, the operations that we wish to perform on values of this type, and our representation for values of this type. Differential propagation, a method of determining and propagating only the bindings of a function that have changed, is described in Section 5. Section 6 shows how we can maintain dependency information for attribute instances computed from finite function values. This information is ordered by the domain value to which the function is applied, so that the attribute instances dependent upon the changed portions of the finite function can be efficiently determined. The complete propagation algorithm, using differential propagation and the ordered dependency information, is discussed in Section 7. In Section 8, we show how symbol tables are implemented using finite functions and we discuss the asymptotic performance of such an implementation.

3.0 Previous Approaches to the Aggregate Problem

There have been numerous attempts to solve the aggregate problem for attribute grammars. In [J84, JF85], attribute grammars are extended to allow nonlocal productions. These nonlocal productions allow nonlocal dependencies to be created between identifiers and their uses distant in the parse tree. Thus, when a change is made to the definition information for an identifier, the

change can be propagated directly to the uses of that identifier. The introduction of additional definitions to the symbol table, however, still requires that all uses of the symbol table be examined in order to update the nonlocal dependencies.

[DRZ85] present a different approach based upon message passing. In their system, a declaration change is an inexpensive operation, but a change to an identifier reference requires a query message to be sent to the symbol table and a reply returned with the updated information. A scheme similar to our differential propagation is used to transmit changes in message sets.

In [BC85], maintained and constructor attributes are used to specify implicit dependencies between identifier declarations and uses, but it is unclear how these implicit dependencies are maintained with respect to arbitrary subtree replacements.

While these approaches reduce the cost of incrementally updating aggregate values after changes, they have all found it necessary to extend the attribute grammar formalism to do so. With the introduction of the finite function type in our specification language, we are able to perform an efficient incremental update after changes to either aggregate value definitions or aggregate value uses. The result is a totally declarative attribute grammar specification.

A similar approach that allows the efficient updating of aggregate values in upward remote references is sketched in [RMT86].

4.0 Finite Functions

Let f be a function from domain D to range R. We call f *finite* if there exists $r_0 \in R$ such that $\{x \in D \mid f(x) \neq r_0\}$ is finite. We denote the type of such a function by $D \rightarrow R[r_0]$ and refer to r_0 as *bottom*.

We represent keyed aggregate values with finite functions. Each element in the aggregate value becomes a pair that maps an element of D, the key, to an element in R, the value being communicated from definition to use.

In his original paper on attribute grammars [K68], Knuth permitted global finite functions, using them to represent symbol tables in an unscoped language. We allow attributes whose values are finite functions. This gives us the capability to represent, among other things, block structured scoping rules.

4.1 The Finite Function Type

We define the following operations for finite functions that map type D to type R.

Declare — An attribute f is declared to have finite function type $D \rightarrow R[r_0]$ by the syntax f : **function** $D \rightarrow R$ **bottom** r_0. In order to permit static typechecking, we will require r_0 to be a constant value in R.

Construct — The expression [**bottom** r_0, $d_1 \leftarrow r_1$, $d_2 \leftarrow r_2$, ..., $d_n \leftarrow r_n$] evaluates to a finite function of type $D \rightarrow R[r_0]$ with domain elements $d_1, d_2, ..., d_n \in D$ bound to the range elements $r_1, r_2, ..., r_n \in R$ and all other domain elements bound to $r_0 \in R$. If the value of r_0 can be determined from context, we will allow it to be omitted. As above, we require r_0 to be a constant value. The values $d_1, ..., d_n, r_1, ... r_n$ are arbitrary expressions.

Apply — Given expression f of type $D \rightarrow R[r_0]$ and expression d of type D, the expression $f(d)$ evaluates to the value $r \in R$ bound to d in the function f.

Update — For two expressions f_1 and f_2 of type $D \rightarrow R[r_0]$, the expression f_1 **update** f_2 results in a new function f of type $D \rightarrow R[r_0]$ defined by $f(d) =$ **if** $f_2(d) \neq r_0$ **then** $f_2(d)$ **else** $f_1(d)$. Thus, the expression f **update** $[d \leftarrow r]$ denotes the function with the same bindings as f except for the value of d bound to r.

Other operations on finite functions can be defined. We will limit our discussion to the above operations as they are sufficient to implement symbol tables in block structured languages. If needed, other operations can be implemented in a similar manner.

We assume that every finite function and every argument to which a finite function is applied is

the value of an associated attribute that is stored in the derivation tree. In order to satisfy this assumption, it may be necessary for the grammar analyzer to introduce new attributes in which to preserve the intermediate values of sub-expressions.

4.2 Finite Function Representation

There are two properties that we would like our representation of finite functions to have. It must be inexpensive to perform the construct, update, and apply operations, and it must be possible to perform these operations without altering the representation of the original finite function–an attribute value that we wish to maintain.

A technique frequently used to implement keyed aggregate values is to store the set as a linked list, inserting each element at the head of the previous list. When an element is needed, a sequential search is performed on the list to determine the first occurrence, if any, of the key in the list. As most incremental attribute grammar evaluators require that intermediate attribute values be maintained, this implementation allows total sharing of data and a unit time insertion operation.

There are two major disadvantages to this technique. First, the key lookup is an expensive O(|*definitions*|) operation. Second, since all uses are functionally dependent upon the aggregate value, any change to the aggregate value will require us to reevaluate all lookup operations to determine if the element for that key has changed. Thus, the change of any element of an aggregate value can take O(|*definitions*|*|*uses*|) time to incrementally update. When using this method to represent the symbol tables of large block structured programs with many identifier definitions and uses, this cost is intolerable.

We use the applicative balanced trees [KM, M83, R84] and represent finite functions as a *map*. A map is a set of n domain-range pairs ordered by domain value. This allows us to insert, remove, and locate definitions in O(log n) time. In addition, it allows all but O(log n) elements of the new tree to be shared with the old tree. Not only does this save storage, but it eliminates the O(n) cost of duplicating the old tree.

This organization of bindings requires us to impose a total order upon the domain D. It is not necessary, however, that this ordering correspond to a logical ordering (i.e. alphabetic) of the domain values. For complicated objects, a scheme such as hash consing [A78], which gives values unique locations and allows unit time comparison, should be used.

We have used the applicative AVL trees [M83] to implement the following primitive operations on maps. The constant *no_pair* refers to the pair **<nil, nil>**.

Lookup(m, d) If there exists a pair in the map m that has domain value d, that pair is returned. If d is not defined in m, *no_pair* is returned.

Delete(m, p) If the pair p is in map m, a map with every pair in m except for p is returned.

Insert(m, p) A map is returned with every pair in the map m plus the pair p. If a pair in m has the same domain value as p, it is replaced in the new map by p.

Note that the delete and insert operations are non-destructive. The map m is not altered to compute the new value.

We also make use of a procedure that combines two maps into a resulting map. This procedure, shown in Figure 1, takes two maps and three functions. These functions return a pair that is to be included in the resulting map or, if no pair is to be included, *no_pair*. If a domain element appears in pairs of both maps, the first procedure is called with both pairs. If the domain element appears only in one of the mappings, the second or third function is called with the pair from the first or second map respectively.

The function ident returns the passed pair and is used for the second or third function when the resulting mapping contains all pairs appearing in the corresponding mapping. When this function is used for a map sufficiently larger than the other map, combine will insert the pairs of the smaller map into the larger map by calling the function tree_insert. The time required by this function is O($|map_{small}|$ log$|map_{large}|$). If the ident

```
function combine(
   m1, m2 : map,
   function in_both(p1,p2 : pair) : pair,
   function in_1(p : pair) : pair,
   function in_2(p : pair) : pair
) : map
   if |m1|>|m2|*log|m1|
   and in_1 = ident then
      r←tree_insert(m1,m2,in_both,in_2,true)
   else if |m2|>|m1|*log|m2|
   and in_2 = ident then
      r←tree_insert(m2,m1,in_both,in_1,false)
   else
      r←tree_merge(m1,m2,in_both,in_1,in_2);
   return(r);

function ident(p : pair) : pair
   return(p);
```

Figure 1

function is not used, or the mappings are roughly the same size, the function tree_merge is used to combine the mappings. This requires $O(|map_1|+|map_2|)$ time. Tree_insert and tree_merge are given in Figures 2 and 3.

```
function tree_insert(
   m1, m2 : map,
   function in_both(p1,p2 : pair) : pair,
   function in_2(p : map) : pair,
   in_order : boolean
) : map
   m←m1;
   for all p2∈m2 do
      p1←lookup(m, p2.domain);
      if p1≠no_pair then
         if in_order then
            add_p←in_both(p1, p2)
         else add_p←in_both(p2, p1)
      else add_p←in_2(p2);
      if add_p≠no_pair then
         m←insert(m, add_p)
      else m←delete(m, p1);
   return(m);
```

Figure 2

4.3 Evaluating Expressions with Finite Functions

To evaluate a Construct expression,we build a map that contains all of the specified bindings, ignoring any bindings to the specified bottom

```
function tree_merge(
   m1, m2 : map,
   function in_both(p1,p2 : pair) : pair,
   function in_1(p : map) : pair,
   function in_2(p : map) : pair
) : map
   p_list = ∅;
   p1←traversal_first(m1);
   p2←traversal_first(m2);
   while p1≠no_pair or p2≠no_pair do
      if p2 = no_pair then
         add_p←in_1(p1);
         p1←traversal_next(m1)
      else if p1 = no_pair then
         add_p←in_2(p2);
         p2←traversal_next(m2)
      else if p1.domain = p2.domain then
         add_p←in_both(p1, p2);
         p1←traversal_next(m1);
         p2←traversal_next(m2);
      else if p1.domain < p2.domain then
         add_p←in_1(p1);
         p1←traversal_next(m1)
      else if p1.domain > p2.domain then
         add_p←in_2(p2);
         p2←traversal_next(m2);
      if add_p≠no_pair then
         p_list←p_list||add_p;
   m←make_tree_from_ordered_list(p_list);
   return(m);
```

Figure 3

value. Since the binding pairs must be ordered, constructing a finite function requires $O(n \log n)$ time where n is the number of bindings.

Applications are evaluated by performing a lookup operation on the map with the given domain value. If there is a pair in the map with the given domain value, its corresponding range value is used as the value of the expression. Otherwise, the bottom value for the map is used. This requires time proportional to the log of the map size. In Section 6 we will show how this lookup is avoided for applications of finite functions that are copied by copy chains, the usual case.

To compute g **update** h, we build a new map of all pairs of g and h using those pairs from h when pairs in g and h have identical domain values. This is achieved by the update function given in

Figure 4. It requires $O(\min(|g|\log|h|, |h|\log|g|, |g|+|h|))$ time to compute the new map.

```
function update(m1, m2 : map) : map
   function in_both(p1,p2 : pair) : pair
      return(p2);
   m←combine(m1,m2,in_both,ident,ident);
   return(m);
```

Figure 4

5.0 Differential Propagation

Most incremental attribute grammar evaluators use what has been called change propagation [Y83, J84, R84, H86]. When the value of an attribute instance a is computed and it differs from the previous value for a, all attribute instances that have values depending on the value of a are required to be reevaluated. With a naive implementation of aggregate values, a change in an aggregate valued attribute requires all uses of that value to be reevaluated, even when the change does not affect the use.

When the value of an attribute instance with finite function value f changes, we wish to propagate the change only to the attribute instances that depend upon bindings of f that have changed. To do this, we need to identify the function bindings that are different. We use differential propagation to create a *delta set*, a set of bindings that have changed between the old and new finite function values of an attribute instance. This is similar to techniques used by [L79, PK82, DRZ85, H85].

The delta set for the change in a finite function of type $D \rightarrow R[r_0]$ from f_{old} to f_{new} is a finite function Δ_f of type $D \rightarrow R \cup \{\textbf{nil}\}[r_0]$ where **nil** is a special value not in R. The bindings for Δ_f are defined as follows.

$<d, f_{new}(d)>$ iff $f_{new}(d) \neq f_{old}(d)$ and $f_{new}(d) \neq r_0$

$<d, \textbf{nil}>$ iff $f_{new}(d) \neq f_{old}(d)$ and $f_{new}(d) = r_0$

$<d, r_0>$ iff $f_{new}(d) = f_{old}(d)$

To compute Δ_f, we can simply compare the bindings of f_{new} with the bindings of f_{old}. The function delta_set, shown in Figure 5, does this. Its operation requires $O(\min(|f_{old}|\log|f_{new}|, |f_{old}|+|f_{new}|))$ time to compute Δ_f where $|f_{old}|$ and $|f_{new}|$ are the number of pairs in f_{old} and f_{new} respectively.

```
function delta_set(m1, m2 : map) : map
   function in_both(p1,p2 : pair) : pair
      if p1.range = p2.range then
         return(no_pair)
      else return(p2);
   function in_1(p : pair) : pair
      return(<p.domain, nil>);
   Δ←combine(m1,m2,in_both,in_1,ident);
   return(Δ);
```

Figure 5

For some updates we can do better than this. For an attribute instance a whose value f is defined to be equal to g **update** h, we can compute Δ_f from Δ_g and Δ_h. This delta_update operation, shown in Figure 6, requires $O(|\Delta_g|\log|h| + |\Delta_h| + |\Delta^{\textbf{nil}}{}_h|\log|g|)$ time, where $|\Delta^{\textbf{nil}}{}_h|$ is the number of elements of D bound to **nil** in Δ_h. f_{new} is then computed from f_{old} and Δ_f in $O(\min(|\Delta_f|\log|f_{old}|, |\Delta_f|+|f_{old}|))$ time by the delta_fix function of Figure 7. Computing Δ_f and f_{new} using Δ_g and Δ_h requires that we have consistent values of g and h. If either g or h is inconsistent with f because of a tree modification, we cannot perform this improved update and we must perform the full comparison.

6.0 Finite Function Dependency Sets

When the value of a given attribute instance changes, we reevaluate all attribute instances that are functionally dependent upon that attribute instance. If the value of the attribute instance is a finite function f, we would like to reevaluate only attribute instances that depend upon the parts of that finite function that have changed. In Section 5, we discussed how to compute the delta set, the set of bindings that have changed. In this section, we describe a dependency set organization that allows us to determine the attribute instances that depend upon the changed bindings.

6.1 Copy Bypass Tree

In [H86] we introduced an algorithm that dynamically bypasses copy rule chains in the attributed tree by forming a tree of nonlocal dependencies. These nonlocal dependencies, called copy bypass dependencies, allow

```
function update_from_delta(
  Δ1,m1,Δ2,m2 : map
) : map
  function in_both(p1,p2 : pair) : pair
    if p2.range = nil then
      return(p1)
    else return(p2);
  function in_1(p : pair) : pair
    if lookup(m2, p.domain) ≠ no_pair then
      return(no_pair)
    else return(p);
  function in_2(p : pair) : pair
    if p.range ≠ nil then
      return(p)
    else
      p1←lookup(m1,p.domain);
      if p1 = no_pair then
        return(<p.domain, nil>)
      else return(p1)
  m←combine(Δ1,Δ2,in_both,in_1,in_2);
  return(m);
```

Figure 6

```
function delta_fix(m1,Δ1 : map) : map
  function in_both(p1,p2 : pair) : pair
    if p2.range = nil then
      return(no_pair)
    else return(p2);
  function in_2(p : pair) : pair
    if p.range = nil then
      return(no_pair)
    else return(p);
  m←combine(m1,Δ1,in_both,ident,in_2);
  return(m);
```

Figure 7

propagation to go directly from an attribute instance to the noncopy attributes that are functionally dependent upon it. The *copy bypass tree*, which contains these dependencies, is ordered by path information so that nonlocal dependencies that cross points of subtree replacement in the attributed tree can be located and removed. Any new copy chains formed by the attributed tree modification are then reinserted into the copy bypass tree during propagation. Figure 8 shows a typical copy bypass tree for an attribute f that is copied through the attributed tree by chains of copy rules. Instances of copy attributes that are nontrivially used are indicated by boxes, and the corresponding noncopy attributes that depend upon them by dots. The copy bypass tree for f contains dependency pointers to all nontrivially used copy attributes, which in turn point back to the definition of f at the head of the tree of copy chains.

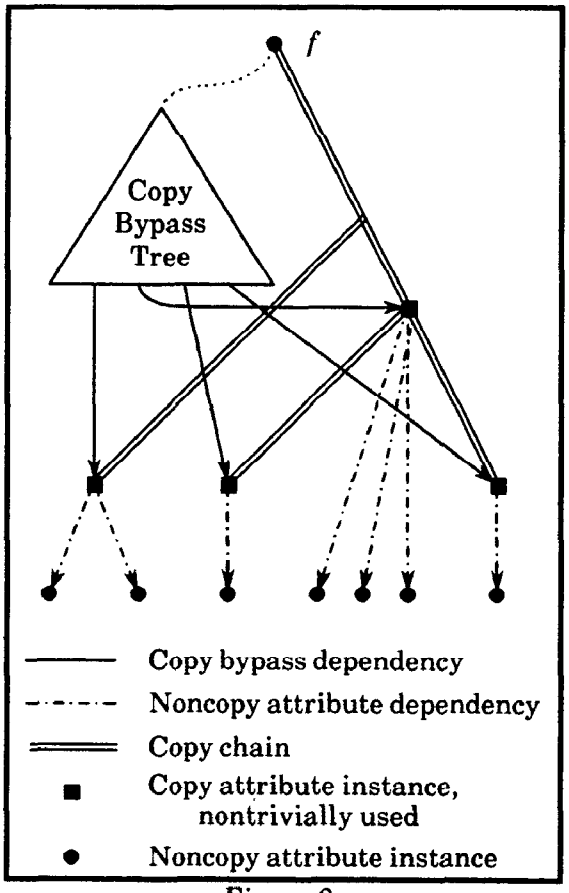

Figure 8

If no copy chains originate from f, the number of uses that depend upon f is limited to the number of semantic equations that refer to f in the two production instances where f is visible. For any given attribute grammar, this is bounded by a (usually small) constant. Thus, we will not attempt to limit propagation of finite function values to attribute instances that locally depend upon f. It is therefore sufficient to consider the set of dependencies that are formed through copy chains. This set is exactly the set of nonlocal dependencies represented by the copy bypass tree.

6.2 Application Tree

Unfortunately, the ordering of the copy bypass tree does not allow us to efficiently locate uses that are affected by a change in a given binding of f. We cannot simply reorder the copy bypass tree

because we need its organization to maintain the nonlocal dependencies. Instead, we maintain an additional balanced tree, the *application tree*, as illustrated in Figure 9. The application tree for finite function f contains one entry for every distinct domain value k to which f is applied, ordered by domain values. The application tree entry for k contains the value $f(k)$ and is the head of a doubly linked list of all attribute instances defined by applications of f to k. In addition, a pointer is maintained from each attribute instance on the doubly linked list for $f(k)$ back to the application tree entry for k (not shown in Figure 9).

Some attribute instances (i.e. those defined by update) will depend upon all elements of the finite function f, and must be reevaluated following any change to f. A separate doubly linked list of these attributes, called the *always propagate list*, is maintained. Whenever any change in the finite function value occurs, all attribute instances on this list are scheduled for evaluation.

6.3 Application Tree Consistency

We must maintain the application tree data structure in the presence of changes to the attributed tree. In Section 6.3.1, we show how the copy bypass tree is used to remove dependencies invalidated by a subtree modification. Section 6.3.2 describes how dependencies are added and updated.

6.3.1 Removing Invalid Dependencies

In [H86], a method is given that locates and removes all copy bypass dependencies invalidated by a subtree replacement. The dependencies represented by the doubly linked lists of the application tree are valid if and only if the corresponding copy bypass dependency is valid. Therefore, at the same time as we remove invalid copy bypass dependencies for f, we can remove all corresponding application dependencies and *always propagate* dependencies. If we remove the last attribute from a given application tree list, we remove the corresponding element from the application tree.

6.3.2 Adding and Correcting Dependencies

Whenever we evaluate an attribute that has a finite function argument, we must insert the attribute being evaluated into the data structures provided it is not already on a doubly linked list. If the attribute is defined by an update operation, we insert it into the *always propagate* list. Otherwise, we locate the application key in the application tree, insert the attribute on the list for that key, and set the pointer to the application tree element. If the key is not in the application tree, we must insert it.

If the attribute is already on a doubly linked list, we verify that it is on the correct one. This must be done since the application key may have been changed by the tree modification. We use the pointer to the application tree element to find the former key. If the key has changed, we must remove the attribute from the doubly linked list and place it on the correct one.

6.4 Time Bounds for Application Tree

We will assume the copy bypass tree. See [H86] for a discussion of its maintenance time.

The use of doubly linked lists allows us to remove most application dependencies in unit time. An $O(\log |\textit{application tree}|)$ operation is required to insert a new dependency or to remove the last dependency for a given key. If the key to an application has not changed, the usual case, we can validate the dependency in constant time. Otherwise, it takes an $O(\log |\textit{application tree}|)$ operation. Since the result value is stored in the application tree, the cost of performing the application is absorbed in the validation cost.

Note that only distinct, used identifiers contribute to the size of the application tree. In a block structured language, the number of distinct identifiers used in a given scope is typically small compared to the number of identifier uses in the scope or the number of definitions in the symbol table. Therefore, the $O(\log |\textit{application tree}|)$ cost of inserting a dependency to a previously unused identifier is relatively inexpensive.

7.0 Finite Function Propagation

To incrementally update an attributed tree after a change, we use the update algorithm of [H86] with the following additions for finite functions.

Before a subtree is replaced, [H86] removes all copy bypass dependencies that would become invalid when the old subtree is separated. As each copy bypass dependency is removed, we remove all of its noncopy dependencies from their respective doubly linked lists as described in Section 6.3.1. The maximum additional time required to remove a subtree is O(|*application tree*|*log |*application tree*|).

The propagate algorithm of [H86] uses change propagation over local and copy bypass dependencies. It starts with a priority queue of inconsistent attribute instances. While the queue is not empty, it removes the first attribute instance and computes its new value. If the new value is different from the old value, it inserts all local and nonlocal successors of the attribute instance into the priority queue.

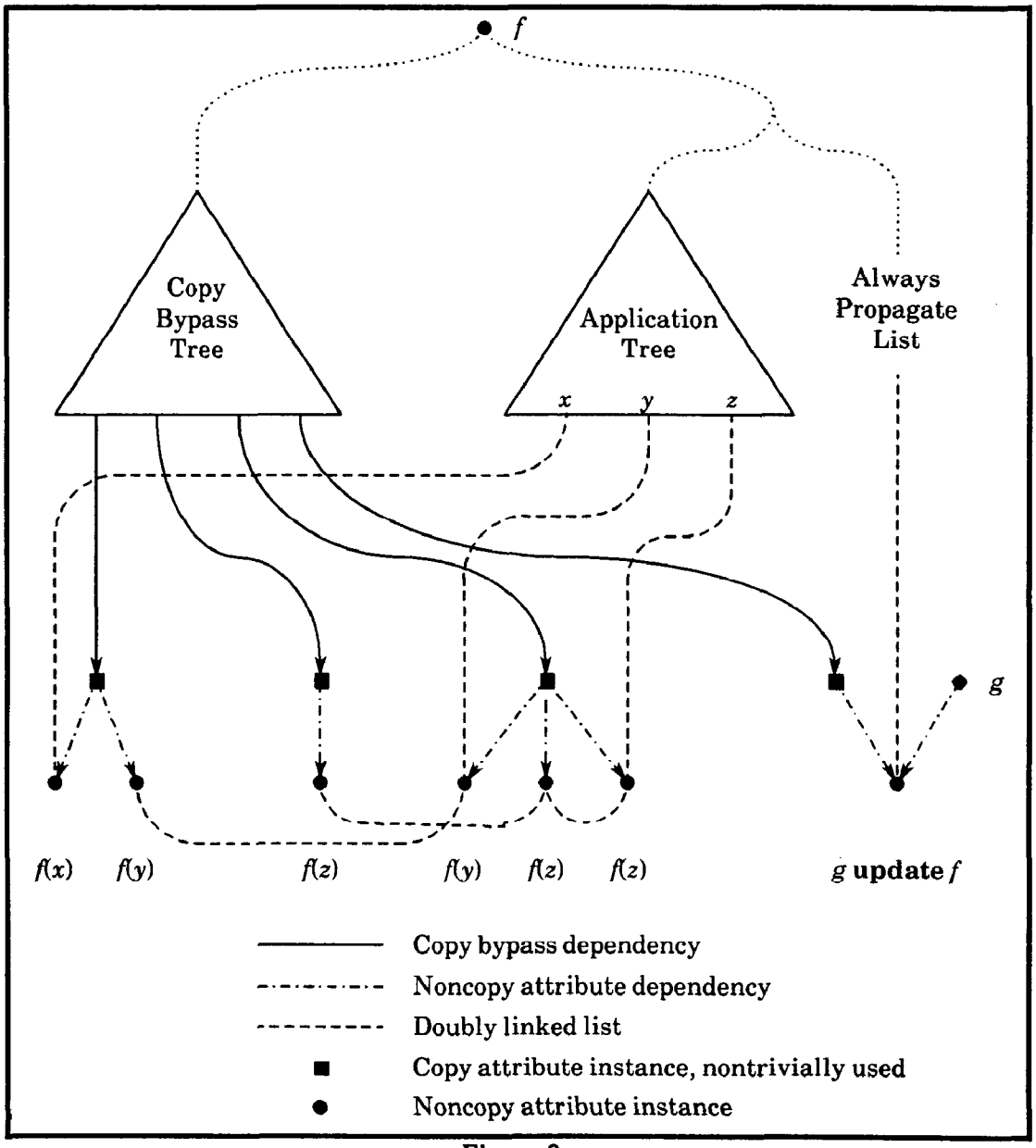

Figure 9

Our propagate algorithm is the same except for each attribute instance that has a finite function value f. To perform differential propagation, we need a value for Δ_f. If f is possibly inconsistent because of the subtree replacement, or f is built by function construction, we compute the new value from its arguments and use the delta_set function of Figure 5 to compute Δ_f. Otherwise, as discussed in Section 5, we have delta sets for either of the arguments of an update expression, and the function update_from_delta in Figure 6 is used to compute the delta set from the delta sets of the arguments.

If the Δ_f has at least one nonbottom binding, we know that the old value is different from the new value and we must enqueue all attribute instances dependent on f. We do this by enqueueing all attribute instances with dependencies on the *always propagate* list and then calling the procedure function_enqueue shown in Figure 10. This limits propagation to the elements of the application tree whose domain values are in the set Δ_f. This procedure requires $O(\min(|\textit{application tree}|\ \log|\Delta_f|, |\Delta_f|\ \log|\textit{application tree}|, |\textit{application tree}|+|\Delta_f|))$ time.

Each attribute instance b that is defined with an update expression using f will be on the *always propagate* list. Before we insert b into the priority queue, we attach the delta set for argument f. When b is evaluated, it will have accumulated the delta sets for all of its changed arguments. When we compute the delta set for b, we remove the delta sets of its arguments.

Since the approximate topological ordering method of [H86] can result in some attribute instances being evaluated out of topological order, we must allow argument delta sets to be updated when the argument is computed more than once. If b already has a delta set for argument f, we need to update this delta set to reflect the changes in both delta sets, old and new. If any bindings are in both sets, we take the new bindings. Thus, the new argument delta set is Δ_{old} **update** Δ_{new} and can be computed using the update algorithm given in Figure 4.

Note that it is possible to organize the application tree as a hash table resulting in a faster expected location time for small changes in f. For efficiency reasons, however, it is necessary that we be able to traverse the hash table in $O(|\textit{application tree}|)$ time.

```
procedure function_enqueue(
   S : priority queue,
   Δ : map,
   t : application_tree
)
   if |Δ| > |t|*log|Δ| then
      for all a∈t do
         p←lookup(Δ, a.domain);
         if p ≠ no_pair then
            appl_enqueue(S, Δ, a, p)
   else if |t| > |Δ|*log|t| then
      for all p∈Δ do
         a←appl_lookup(t, p.domain);
         if a ≠ no_appl then
            appl_enqueue(S, Δ, a, p)
   else
      a←traversal_first(t);
      p←traversal_first(Δ);
      while a ≠ no_appl and p ≠ no_pair do
         if a.domain < p.domain then
            a←traversal_next(t)
         else if a.domain > p.domain then
            p←traversal_next(Δ)
         else if a.domain = p.domain then
            appl_enqueue(S, Δ, a, p);
            a←traversal_next(t);
            p←traversal_next(Δ);

procedure appl_enqueue(
   S : priority queue,
   Δ : map,
   a : appl_dependency,
   p : pair
)
   if p.range = nil then
      a.range←Δ.bottom
   else a.range←p.range;
   for all l∈a.list do
      S←S∪{a.attribute_instance};
```

Figure 10

8.0 Symbol Tables Using Finite Functions

We modified our attribute grammar specification for Pascal to use attributes of finite function type identifier→scope depth × value × kind × type to represent the symbol table.

The initial table is created using a function construction expression with all identifiers

initially defined in Pascal. This finite functional value is inherited by copy chains to the first nested scope.

At each scope, definitions are collected using a local environment, also represented by a finite function. The complete local environment is synthesized from the bottom of the local definition subtree back up to the block head where the new environment is created with the update operator.

Before we insert an identifier definition into the local scope, we must verify that it has not already been defined in the scope. We perform an application of the local environment to the identifier, and, if it is not defined, we construct a finite function with the identifier binding and form the new local environment with the update operator.

8.1 Asymptotic Performance of Symbol Table Modifications

In the following discussion, we will ignore the cost of maintaining nonlocal dependency information with respect to abstract syntax tree modifications. As is discussed in Section 6, this is done using copy bypass dependencies. Since the application dependencies can be maintained in the same time bound as the copy bypass dependencies, we refer to [H86] for an analysis of this cost.

Let us assume that we are modifying the symbol table entry for an identifier *id* that has u uses in each of the s enclosed scopes where it is visible. Let l be the number of definitions local to the scope of modification and let d be the number of definitions in the largest enclosed scope. We will assume that no other definitions are dependent upon *id*. The type of modification is not limited to changing the definition information. We consider inserting and removing definitions as well.

When we change the definition of *id*, we must update each of l partial local environments. Computing the delta set with update_from_delta (Figure 6), and then computing the new value with delta_fix (Figure 7) requires $O(\log l)$ cost for each local scope. Thus, the cost to update the local definitions is $O(l \log l)$.

At the block head of each modified scope, we require $O(\log d)$ operations to compute the delta set and update the attribute value. The total cost to update the scopes is $O(s \log d)$.

Each of the uses of *id* must also be located and updated. For a given scope, locating the linked list of the uses of the changed definition in the application tree requires $O(\log |\textit{application tree}|)$ time. Subsequently, finding the new value bound to *id* requires unit time for each use. The total time required for locating and updating uses is therefore $O(s^*(u + \log |\textit{application tree}|))$.

If we assume that there are few undeclared identifiers, $|\textit{application tree}| < d$. This gives us the resulting cost of $O(l \log l + s \log d + s\ u)$ for the modification of one symbol table identifier.

As discussed in Section 6, modifying an identifier use, although usually done in unit time, requires at most $O(\log |\textit{application tree}|)$ operations.

9.0 Summary

The incremental update of aggregate values has been a major performance problem in attribute grammar based systems. While there have been numerous attempts to eliminate this bottleneck, all have found it necessary to extend the attribute grammar formalism.

We have introduced the finite function data type to represent these aggregate values. We determine the portion of the aggregate value that has changed using differential propagation. By keeping a tree of aggregate value uses at points in the attributed tree, we are able to locate the attribute instances that depend upon the changed portion of the aggregate value. This allows us to limit propagation to the attribute instances that are uses of changed definitions.

Finite functions allow symbol tables to be represented easily and efficiently in attribute grammars.

10.0 References

[A78] Allen, J. R. *Anatomy of LISP*. McGraw-Hill, New York, NY, 1978.

[BC85] Beshers, G., and R. Campbell. Maintained and Constructor Attributes. *Proc. ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle, WA, June, 1985, pp. 34-42.

[DRT81] Demers, A., T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. *Proc. of the 8th ACM Symposium on Principles of Programming Languages*, Jan, 1981, pp. 105-116.

[DRZ85] Demers, A., A. Rogers, and F. K. Zadeck. Attribute propagation by message passing. *Proc. ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle, WA, June, 1985, pp. 43-59.

[H85] Horwitz, S. Generating Language-Based Editors: A Relationally-Attributed Approach. TR 85-696 (*Ph.D. Thesis*), Cornell University, August 1985.

[H86] Hoover, R. Dynamically bypassing copy rule chains in attribute grammars. *Proc. of the 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, Jan 13-15, 1986, pp. 14-25.

[J84] Johnson, G. F. An approach to incremental semantics. TR 547 (*Ph.D. Thesis*), University of Wisconsin, Madison, July 1984.

[JF85] Johnson, G. F., and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans, LA, Jan 14-16, 1985, pp. 141-151.

[K68] Knuth, D. E. Semantics of context-free languages. *Mathematical Systems Theory*, 2, 2, June 1968, pp. 127-145.

[KM] Krijnen, T. and L. Meertens. Making B-trees work for B. Technical Report, Mathematical Center, Amsterdam.

[L79] Liu, L. Essential uses of expressions in set-oriented expressions. *Ph.D. Thesis*, Cornell University, May 1979.

[M83] Myers, E. W. Efficient Applicative Data Types. *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, Jan, 1983, pp. 66-75.

[PK82] Paige, R. and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 402-454.

[R84] Reps, T. Generating Language-based Environments. M.I.T. Press, Cambridge, MA, 1984.

[RMT86] Reps, T., C. Marceau, T. Teitelbaum. Remote attribute updating for language-based editors. *Proc. of the 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, Jan 13-15, 1986, pp. 1-13.

[RT84] Reps, T. and T. Teitelbaum. The Synthesizer Generator. *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Y83] Yeh, D. On Incremental Evaluation of Ordered Attributed Grammars. *BIT 23*, 1983, pp. 308-320.