Incremental Graph Evaluation

Roger Hoover

PhD Thesis

87-836
May 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

INCREMENTAL GRAPH EVALUATION

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Roger Hoover
May 1987

INCREMENTAL GRAPH EVALUATION

Roger Hoover, Ph.D.
Cornell University 1987

There are many computer applications that can be made incremental. After a small perturbation to the computation at hand, intermediate values of a previous evaluation can be used to obtain the result of the new computation. This requires less time then reevaluating the entire computation.

We propose the use of a directed graph to represent computations that we wish to make incremental. This graph, called a *dependency graph*, represents an intermediate computation at each vertex. Edges between vertices represent the dependence of intermediate computations on other intermediate computations. A change to the computation can be represented as a change in the dependency graph.

Given a dependency graph and a dependency graph modification, we wish to update all intermediate computations to be consistent with the intermediate computations on which they depend. This thesis is a study of the updating process, known as *incremental graph evaluation*.

Graph propagation is a simple technique that restores a modified dependency graph to consistency. We briefly review a number of propagation algorithms that have been used for incremental evaluation and introduce several new graph propagation algorithms.

We show that by performing graph propagation on an improved, but equivalent, dependency graph, incremental graph propagation can be accelerated. The improvement is obtained by replacing indirect communication in the dependency graph with direct communication. Since the indirect communication is inherent in many dependency graphs, the time required by incremental graph evaluation can be decreased substantially.

We present algorithms to incrementally maintain a data structure, called a *structure tree*, which we use to explicitly represent the indirect communication. We apply the structure tree algorithms to explicitly represent two types of indirect communication, copy rule chains and aggregates, which frequently occur in dependency graphs. The

Biographical Sketch

Roger Scott Hoover was born on 24 May 1962, which happened to be the same day that astronaut Scott Carpenter became the second American to orbit the earth. "Roger, Scott," crackled ground control on the radio, and Roger Scott was his name. He received an A.B. degree in Mathematics from Wabash College in May 1982 and a Master of Science degree in Computer Science from Cornell University in May 1985.

Table of Contents

List of Algorithms

List of Figures

1. Introduction

In May 1978, Tim Teitelbaum, then an assistant professor at Cornell University, and employee Thomas Reps began writing a computer program that they hoped would free programmers from some of the mundane aspects of computer programming. The result, called the Cornell Program Synthesizer [Teitelbaum and Reps 81], was a programming environment that combined a syntax directed editor, a pseudocode compiler, and an interactive debugger for PL/CS, a dialect of PL/I used for instruction at Cornell.

Meanwhile, that same year, 350 miles to the east in Cambridge, Massachusetts, Harvard M.B.A. student Dan Bricklin, his partner Bob Frankston, and employee Steve Lawrence were building the Visible Calculator. Bricklin envisioned an electronic blackboard with electronic chalk that would take the drudgery out of business calculations. Their program, VisiCalc, was the first of what are now known as spreadsheets [Kay 84].

A prototype of the Program Synthesizer was demonstrable by December 1978, and it was used to introduce students to computer programming beginning in June 1979. Within two years, it was adopted for programming instruction at Rutgers University, Princeton University, and Hamilton College.

VisiCalc was first sold in January 1979. It began to catch the eye of the industry when it was introduced at the West Coast Computer Faire in May 1979. By 1983, it had sold over a half of a million copies at a list price of $300. Dozens of other software companies began to write and sell spreadsheets.

While the problems these programs solved were totally different, the Cornell Program Synthesizer and VisiCalc had almost everything else in common. They were both written to be run on single user microcomputers. Exploiting the ability to quickly update the information on the computer's display, both programs allow the user to interactively construct and then modify an object, giving the user instant results computed from the current state of the object. In addition, both Visicalc and the original Cornell Program Synthesizer suffered from the same problem—the algorithms used did not scale up.

Upon every modification to a declaration or label in the PL/CS program that was being edited, the Program Synthesizer traversed the entire program tree and recomputed the desired results. This included the generation of pseudocode, and a check for static computation at the vertex. This graph is called a *dependency graph*, and is discussed in Chapter 2.

Given an algorithm that we wish to make incremental, we must be able to translate it into a dependency graph. For spreadsheets, the translation is obvious. Each cell of the grid becomes a vertex in the dependency graph. The computation done at the vertex is the evaluation of the cell formula. The edges connect the cell to other cells referenced in the formula. In choosing a formalism for implementing the successor to the Cornell Program Synthesizer, its authors observed that attribute grammars [Knuth 68] perform a similar translation for computations done on tree structured objects [Demers, Reps, and Teitelbaum 81].

We propose the use of the dependency graph as a framework for incremental computation. Given a dependency graph and a modification to that graph, we need to compute the new values specified by the resulting graph. This dissertation is a study of efficient incremental evaluation of dependency graphs. We believe that the algorithms presented here will lead to many practical incremental systems.

A technique called *graph propagation* allows the reuse of intermediate computations that are not affected by the modification. In Chapter 3, we review graph propagation algorithms that have been proposed by others, and introduce several new algorithms: the topological sorting of vertices dependent upon the modification, explicitly maintaining the topological order of the graph, and approximate topological ordering.

One of the major results in the area of incremental evaluation is the algorithm of [Reps 84]. The propagation algorithm, applicable to the dependency graphs created by any noncircular attribute grammar, is optimal in the sense that it requires bookkeeping time proportional to the number of vertices that change in value. However, many of the values that change are used only for communication and have no bearing on the result of the computation. The computation of such values can be avoided while producing the same results. In Chapter 4, we discuss the assumptions that are required to call the algorithm of [Reps 84] optimal. We relax these assumptions, and show how the performance of graph evaluation can be improved.

Chapter 5 introduces a data structure called the *structure tree*. Structure trees allow transitive dependencies to be easily and efficiently represented in the dependency graph. We give algorithms for maintaining these structure trees in the presence of changes to the dependency graph.

2. Graph Evaluation

2.1 The Dependency Graph

We are interested in evaluating an expression represented by a directed computation graph. The vertices of the graph represent intermediate computations and each vertex is associated with a function and a value. We draw edges between vertices of the graph to represent the functional dependence of the vertex value at the directed end of the edge on the vertex value at the undirected end of the edge.

Definition 2.1.1:

A *dependency graph* is denoted

$D=(V, E, R)$

where

V is a finite set of vertices,

E is a finite set of directed edges (v_1, v_2) where $v_1, v_2 \in V$,

$R \subseteq V$ is the set of *result vertices.*

Each vertex $v \in V$ has an associated *vertex function* f_v whose arity is the indegree of v. Vertices with constant values are defined by 0-ary functions.

As we are frequently unconcerned with R, at times we will refer to a dependency graph as a directed graph (V, E).

Definition 2.1.2:

Let $D=(V, E)$ be a dependency graph, and let v be an element of V. If v has an associated *vertex value* (denoted by val_v), we call v an *attributed* vertex. Otherwise, v is *unattributed.*

Let i be the indegree of v, and let pred[0], pred[1], ..., pred[i-1] be the direct predecessors of v in D in some defined order. If pred[0], ..., pred[i-1], and v are attributed, and $\mathrm{val}_v = f_v(\mathrm{val}_{\mathrm{pred}[0]}, \ldots, \mathrm{val}_{\mathrm{pred}[i-1]})$, then v is *consistent.* Otherwise, v is *inconsistent.*

Definition 2.1.3:

Let $D=(V, E, R)$ be a dependency graph. If all of the vertices in V are consistent, D is a *consistent dependency graph.* The set $\{\mathrm{val}_v \mid v \in V\}$ for a consistent dependency graph D is known as a *solution* of D. The set $\{\mathrm{val}_v \mid v \in R\}$ is called a *result* of D.

need to find a set of vertex values corresponding to the set of result vertices such that these values are part of a solution of D. Typically the dependency graph has null or nonexistent vertex values and we must compute a result of D.

Definition 2.2.1:

> The process of obtaining a result of a given dependency graph is called *graph evaluation.* We refer to an algorithm that performs this process as a *graph evaluator.*

While there are numerous approaches to graph evaluation, the basic idea of each is the same. The dependency graph is traversed in some topological order, and the vertex functions are evaluated to compute the vertex values. Except for values associated with result vertices, a vertex value needs to be retained only until all vertex functions that refer to the value have been computed. The vertex values corresponding to the result vertices compose the output of the algorithm.

If D is acyclic, graph evaluation can be done with a topological sort [Knuth 73], requiring all vertex functions to be evaluated in addition to bookkeeping work proportional to the number of vertices and edges in D. If the dependency graph contains unnecessary computation and vertex functions that are not strict, this evaluation time can be reduced by performing lazy evaluation from the result vertices [Jalili 85]. We evaluate the value of each vertex beginning from each result vertex, demanding the value of each argument as it is needed. The demand initiates the computation of the argument vertex value and the process continues until a vertex is reached whose function uses no arguments. The computations are stored at their corresponding vertices so that they may be reused if encountered as arguments of some other vertex function. Thus, each vertex function is evaluated at most once. Since this process is essentially a depth first search over the graph with reversed edges, the bookkeeping is $O(|\text{vertices}| + |\text{edges}|)$.

If D contains cycles, obtaining values for the vertices that make up the cycle requires a fixed point calculation. Typically, one assigns some least element to each vertex on the cycle and requires the values of these vertices to reach a fixed point after iteration. This evaluation can be done by building a *condensed graph* which consists of a vertex for each strongly connected component of D. An acyclic graph evaluation algorithm is then used on this condensed graph [Jones and Simon 86]. The evaluation of each vertex is a single function evaluation if the vertex corresponds to a single vertex in D.

We refer to the process of changing D into D' as a *dependency graph modification*. Each of the operations that make up the modification is known as a *modification step*. We assume that the appropriate vertex functions are changed to reflect edge changes so that D' is a valid dependency graph.

2.4 Incremental Graph Evaluation

We use dependency graphs to represent computations that we wish to make incremental. Given an initial dependency graph, we compute its corresponding consistent dependency graph. We then wish to perform a sequence of dependency graph modifications and obtain the result for each modified graph.

Definition 2.4.1:

Let D_0 be a consistent dependency graph and let $M_1, M_2, \ldots, M_n$ be a sequence of dependency graph modifications where M_i alters D_{i-1}, forming D_i. We call the process of determining the result for each D_i *incremental graph evaluation* and refer to an algorithm that performs this process as an *incremental graph evaluator*.

Thus, the incremental computation process commences with the evaluation of the initial dependency graph by a non-incremental graph evaluator. After each dependency graph modification, consistency is restored by invoking an incremental graph evaluator.

2.5 Graph Evaluable Schemes

In order to use an incremental graph evaluator to perform computations on an object that is being modified, methods are needed to translate the computation on the object into a dependency graph and to translate modifications to the object into dependency graph modifications.

Definition 2.5.1:

Let B be a set of objects. A system that translates a computation on any $b \in B$ into a dependency graph and also translates modifications to b into dependency graph modifications is known as a *graph evaluable scheme* for the objects in B.

Definition 2.5.2:

Let S be a graph evaluable scheme for a set of objects B. S is said to be *bounded* if there is a constant c such that for every object $b \in B$, the indegree and

$S \rightarrow E$
$S.v = E.v$
$E.s = 0$
$S.e = E.e$

$E_1 \rightarrow E_2 + E_3$
$E_1.v = E_2.v + E_3.v$
$E_2.s = E_1.s$
$E_3.s = E_2.e$
$E_1.e = E_3.e$

$E_1 \rightarrow E_2 * E_3$
$E_1.v = E_2.v * E_3.v$
$E_2.s = E_1.s$
$E_3.s = E_2.e$
$E_1.e = E_3.e$

$E \rightarrow 17$
$E.v = 17$
$E.e = E.s + 1$

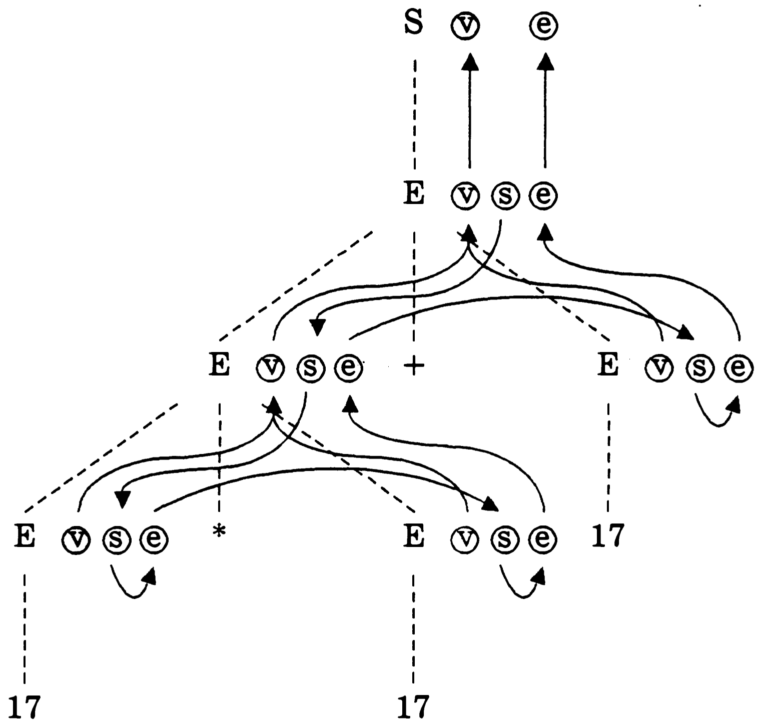


Figure 2.5.1

many graph evaluable schemes, only a small fraction of the total computation is changed by a small perturbation to the object. In the dependency graph, this means that a large portion of the vertex value set for the old graph is the same as the solution of the modified graph. In order to compute the new result of the dependency graph, we save and reuse the intermediate computations from the previous dependency graph.

We characterize the amount of work required to convert the modified dependency graph into the corresponding consistent dependency graph as follows.

Definition 2.6.1:

Let D be a possibly inconsistent dependency graph. The set *AFFECTED* of D is the set of vertices in D whose values in D are different from the values in a solution of D.

Definition 2.6.2:

Let D be the a dependency graph with vertex subset S. *EVAL*(S) is defined to be the total time required to evaluate the vertex function of each element of S using values of the solution of D as the function arguments.

3. Graph Propagation

3.1 Introduction

If we modify a consistent dependency graph using the operations of Figure 2.3.1, the resulting dependency graph may be inconsistent. The set of potentially inconsistent vertices comprises all vertices that have been added, have had one or more incoming edges added or removed, or have had their vertex functions altered. As only a portion of the dependency graph has changed, all other vertices have values that are consistent with their vertex functions.

As we modify the dependency graph, we build a set of the vertices whose values are no longer known to be consistent with their respective vertex functions. We assume that all vertices that have been added to the dependency graph are either consistent, are in this set of possibly inconsistent vertices, or have null values with at least one predecessor with null or inconsistent value. We will also assume that the resulting dependency graph has a solution.

To reestablish the consistency of the dependency graph, we use a graph propagation algorithm. The input to such an algorithm is a dependency graph and a list of possibly inconsistent vertices. The algorithm terminates with a consistently attributed dependency graph.

Numerous propagation algorithms have been proposed for and used in incremental systems. Some of these algorithms can be used for arbitrary dependency graphs, the rest are restricted to a subclass of dependency graphs. In this chapter, we briefly review several of these previously proposed graph propagation algorithms, and introduce several new techniques. These include topologically sorting the vertices dependent upon the graph modifications (Section 3.5), maintaining the topological order of the graph (Section 3.8) and approximate topological ordering (Section 3.10). The last algorithm is treated in more detail in Chapter 8.

3.2 How Graph Propagation Works

We shall assume that we are propagating over a dependency graph $D = (V, E)$ where MODIFIED is the initial set of possibly inconsistent vertices, $n = |V|$, and AFFECTED and EVAL are as defined in Section 2.6. We also make the following definitions.

Definition 3.2.1:

We define *INFLUENCED* to be the set of vertices that either are in AFFECTED, Graph propagation uses these properties as follows. Let W be the vertices in MODIFIED ordered so that they respect a topological order of D. By Lemma 3.2.2, the first of these vertices, v, is guaranteed to have predecessors that are not in AFFECTED. We evaluate the vertex function for v. This value must be consistent with the solution of D.

If the value is equal to val_v, it is not in AFFECTED. Otherwise, if the values are not equal, we update val_v to have the new value and add the direct successors of v to W as they are potentially inconsistent. In either case, we remove v as it is known to be consistent with the solution of D.

We continue this process until no vertices are known to be possibly inconsistent. Lemma 3.2.1 guarantees that all vertices are consistent at termination as MODIFIED contained all vertices that were initially inconsistent, and all vertices that were direct successors of vertices in AFFECTED were verified to be consistent.

3.3 A Lower Bound for Graph Propagation

The above process examines all vertices in INFLUENCED. At each vertex, the vertex function is evaluated and the resulting value is compared with the vertex value. If we assume that we must evaluate the vertex function and compare the values to determine if the vertex is in AFFECTED, and we assume that we can test the equality of two vertex values in less time than is required to evaluate the vertex functions that built them, graph propagation is obviously $\Omega(|\text{INFLUENCED}| + \text{EVAL}(\text{INFLUENCED}))$.

Let D be a graph with vertex subset W. If we assume a process that can determine, in constant time, the first element of W in some topological order of D, we can construct a graph propagation procedure that requires $O(|\text{INFLUENCED}| + \text{EVAL}(\text{INFLUENCED}))$ time. This procedure, shown in Algorithm 3.3.1, is invoked with an inconsistent dependency graph and the set of modified vertices.

In the following sections, we discuss various algorithms that perform the process of finding a vertex in W without predecessor in W. For some subclasses of dependency graphs and dependency graph modifications, these algorithms perform the task in constant time per request. It is an open problem if this time bound can be reached for any dependency graph modification on any given dependency graph.

behavior can occur whenever we remove a vertex from W that has a predecessor in W and can cascade to make the number propagation steps, vertex function evaluations, and vertex value comparisons exponential in the size of DEPENDENT [Reps 84].

3.5 Topological Sort of DEPENDENT

Another strategy for updating the inconsistent dependency graph is to topologically sort the vertices of the graph. Since AFFECTED is a subset of DEPENDENT, the topological sort can be restricted to the vertices in DEPENDENT. The method is an improvement on the two phase nullification and reevaluation algorithm presented in [Demers, Reps, and Teitelbaum 81] and [Reps 84] and is similar to the demand evaluation method of [Hudson 86].

Topological sorting is a two phase process. In the first phase, we start with the vertices in MODIFIED and pass through all transitive successors marking each vertex. Let DE be the set of edges between vertices in DEPENDENT (that is, $DE = \{(v_1, v_2) \in E \mid v_1, v_2 \in \text{DEPENDENT}\}$). For unbounded dependency graphs, an integer field in each vertex is initialized to be the number of predecessors. The first phase is done with a depth first search in $O(|\text{DEPENDENT}| + |\text{DE}|)$ time.

In the second phase, we construct a set S of vertices in MODIFIED that have no marked predecessors. We use this set as our initial set of possibly inconsistent vertices. In addition to the normal steps of graph propagation, we also unmark the vertex and if its value does not change, we traverse its dependent vertices in depth first order unmarking them. For unbounded dependency graphs, the integer vertex field is updated to be the number of unmarked predecessors. We back out of the depth first search when we reach a vertex that has a marked predecessor. This phase requires $O(|\text{DEPENDENT}| + |\text{DE}| + \text{EVAL(INFLUENCED)})$ time.

The total time required by the graph propagation algorithm is therefore $O(|\text{DEPENDENT}| + |\text{DE}| + \text{EVAL(INFLUENCED)})$. For bounded dependency graphs, this is $O(|\text{DEPENDENT}| + \text{EVAL(INFLUENCED)})$. If the size of DEPENDENT is $O(|\text{INFLUENCED}|)$, this algorithm is optimal.

3.6 Optimal Graph Propagation for Attribute Grammars

We can perform optimal graph propagation if we restrict ourselves to dependency graphs created by attribute grammars and restrict the types of graph changes allowed. The algorithm of [Reps 82] [Reps, Teitelbaum, and Demers 83] [Reps 84] specify the topological ordering of the dependency graph vertices. As a plan executes, it visits the vertices in topological order, calling other plans at the necessary times.

The construction of these plans for a large class of attribute grammars, called *absolutely noncircular attribute grammars*, was shown by [Kennedy and Warren 76]. A subclass of these attribute grammars that have simpler linear plans called *visit-sequences* are known as *ordered attribute grammars* [Kastens 80]. The construction of plans for numerous other attribute grammar subclasses has been discussed as well [Warren 75] [Kennedy and Warren 76] [Warren 76] [Cohen and Harry 79].

If we limit our graph modifications to those corresponding to a subtree replacement, we can derive the projection of D on the vertices in the initial set W as was done in Section 3.6, and propagate in topological order using the plans for these vertices. The necessary construction for absolutely noncircular attribute grammars was given by [Reps 84]. The simpler case of ordered attribute grammars can be found in [Yeh 83].

3.8 Maintained Topological Ordering

It is also possible to incrementally maintain a true topological order of the vertices in the dependency graph. Initially, the dependency graph is topologically sorted and placed into a list in topological order. As we perform graph modification steps on the dependency graph, we incrementally adjust the order of this list to reflect the changes. When the dependency graph modification is complete, the order of the list will be a topological order of the dependency graph. We will use the topological order of the vertices in this list to schedule the vertex functions for evaluation during graph propagation.

We use the result of [Tsakalidis 84] [Dietz and Sleator 87] to store the ordered vertices. These algorithms construct a *generalized linked list*, which is a linear linked ordering of the items in the list that has the following properties.

- Removing an item from the list requires O(1) amortized time.
- Inserting an item before or after another item already in the list requires O(1) amortized time.
- Queries about the relative order of two items in the list take O(1) worst case time.

As we perform the initial evaluation of the dependency graph in topological order, we place the vertices into the generalized linked list in the evaluation order. Since the

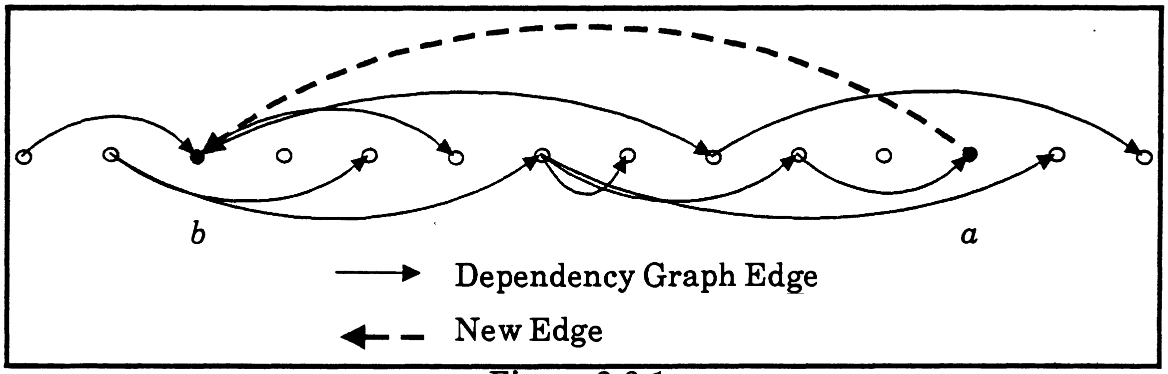


Figure 3.8.1

We extend the dependency graph representation to explicitly represent the predecessor dependencies and use the *reorder* procedure shown in Algorithm 3.8.1. We call this procedure with the source and destination of each edge that we wish to add to the dependency graph. The procedure terminates with the destination vertex to the right of the source vertex, allowing the edge to be added pointing rightward.

The *reorder* procedure searches leftward from the source and rightward from the destination in order of the linked list until a predecessor *p* of the source is found which is right of a successor *s* of the destination. A list is kept of all vertices traversed during this process, and they are removed from the linked list and inserted after *p* and before *s* respectively. This places the source before the destination in the generalized linked list.

We perform the search leftward from the source using a priority queue of predecessor vertices ordered by their relative positions in the generalized linked list. Since the next predecessor to the left must be the right most vertex in the priority queue, the delete_max operation on the priority queue finds this vertex in log |vertex queue| time. Likewise, a priority queue in the reverse order can be used to locate vertices for the search rightward from the destination.

Should we reach a vertex *v* that is both a predecessor of the source and a successor of the destination, inserting the new edge will create a cycle in the dependency graph. If the graph evaluable scheme in use can create cycles, we must check for this condition and if it occurs, we must condense the cycle into a single vertex of the dependency graph as discussed in Section 2.2.

After the graph modification is complete, we perform graph propagation to return the dependency graph to consistency. Since we have a topological ordering of the modified dependency graph, we can replace the process of finding the first vertex in W of Section 3.3 with a priority queue. By placing the vertices of the possibly inconsistent set W into the priority queue ordered by the relative generalized linked list order, we can determine the next vertex to evaluate in O(log|INFLUENCED|) time. The number of operations that are required to return the dependency graph to a consistent state is O(|INFLUENCED|*log|INFLUENCED| + EVAL(INFLUENCED)). *Propagate_graph* is shown in Algorithm 3.8.2.

```
propagate_graph(D : dependency graph, W : vertex priority queue)
        while W ≠ ∅ do
                v←delete_min(W);
                newvalue←f_v(values of pred(v));
                if val_v ≠ newvalue then
                        val_v←newvalue;
                        W←W∪succ(v)
```

Algorithm 3.8.2

3.9 Static Priority Numbering

The dependency graphs generated by some graph evaluable schemes can be propagated using a static priority system. The vertices of the dependency graph are partitioned into a finite collection of sets. Each set is assigned a priority number. Let u be a vertex that is transitively dependent upon another vertex v. If u and v could appear in the possibly inconsistent set of vertices at the same time, the partition is constructed so that u and v will be in different sets. In addition, the priority number for the set containing v will be less than the priority number for the set containing u.

Given the priority numbers for each vertex, it is simple to force evaluation to proceed in topological order. A priority queue is used to arrange the possibly inconsistent set of vertices in the order of their sets. Vertices in the set with the lowest number are evaluated first. The same propagation algorithm used for maintained topological ordering, shown in Algorithm 3.8.2, can be used.

To determine the sets and their priorities, computation is done upon the graph evaluable scheme that generates the dependency graphs. [Johnson and Fischer 85] compute static priority numberings for a subclass of attribute grammars. A similar

4. Improving Incremental Graph Evaluation

4.1 Why Optimal Graph Propagation Does Not Result in Optimal Incremental Graph Evaluation

In the previous chapter, we derived a lower bound on graph propagation of $\Omega(|\text{INFLUENCED}| + \text{EVAL}(\text{INFLUENCED}))$, and claimed that any propagation algorithm that runs within this bound is optimal for incremental evaluation. We made some assumptions, however, that when relaxed, open the possibility for improvement. We discuss these assumptions below.

4.1.1 On-line Algorithm Assumption

In Definition 2.4.1, we defined an incremental graph evaluator to be an algorithm that updates a dependency graph after a single dependency graph modification. This implies the incremental evaluation is done *on-line* [Aho, Hopcroft, and Ullman 74], the incremental graph evaluator performing each update without knowledge of modifications later in the modification sequence.

It is possible to improve the performance of the incremental evaluation process if an on-line algorithm is not required. For example, if modification M_i performed the inverse of modification M_{i+1}, the algorithm could remove them from the modification sequence yielding identical results at the end of the sequence.

Most uses of incremental evaluation, however, require the intermediate dependency graphs after each modification and before the next modification. If this is not the case, we will combine with M_{i+1} any dependency graph modification M_i whose corresponding intermediate dependency graph D_i is not needed, performing both modifications at once. In either case, we will continue to make the on-line algorithm assumption.

4.1.2 Equality Test Assumption

We also assumed that we could test the equality of vertex values and that the test could be done in less time than is required to compute them from their vertex functions. If we cannot test for equality, we cannot do propagation and must resort to the partial topological sorting method of Section 3.5, performing no comparisons. The resulting algorithm evaluates all vertices in DEPENDENT.

If testing for equality takes more time than recomputing the value, propagation could do far worse than the above DEPENDENT evaluation method. To limit the computation

4.2 Optimal Incremental Graph Evaluation

Since optimal graph propagation does not yield optimal incremental graph evaluation, we would like to characterize what optimal incremental graph evaluation is. Although it is unachievable, we will use the optimality concept to motivate our efforts to improve the performance of incremental graph evaluation.

In the case of non-incremental graph evaluation, we can think of the computation represented by the dependency graph as one large function that computes a set of values, one for each result vertex of the dependency graph. In other words, a non-incremental graph evaluator is a function of type Dependency Graph $\rightarrow$ Result. Hence, an optimal graph evaluator is an optimal program for that function.

In the incremental case, the evaluator becomes a function of type Dependency Graph $\times$ Modification Sequence $\rightarrow$ Result Sequence where the first element of the result sequence is the result of the original dependency graph and each following element corresponds to the result after each modification. In addition, we assume that the result sequence is produced on-line as discussed in Section 4.1.1. An optimal incremental evaluator, therefore, is the optimal on-line program for this function.

As the automatic construction of the above optimal programs is undecidable, its discussion is well beyond the scope of this thesis. However, as was the case with graph propagation, it is useful to consider graph evaluators that are optimal given a set of restrictions upon the problem. In the remainder of this chapter we weaken the assumptions of Section 4.1 and develop a concept of optimality given these assumptions. We argue that these weaker assumptions are more realistic than the assumptions of Section 4.1.

4.3 More Reasonable Assumptions for Graph Evaluation

In our attempt to improve the performance of incremental evaluation, we are faced with the following dilemma. On the one hand, as asserted in Section 4.1, there is the possibility for improvement if we transform the dependency graph into a dependency graph better suited for incremental evaluation. On the other hand, the generality of our model of computation—the dependency graph and the incremental modifications allowed—make this task difficult.

evaluation—the edges of the dependency graph may be changed to provide better communication of values between vertices.

4.4.1 Implicit Identity Dependencies

While we are unable to change the vertex function at vertex v, we can change the vertices where the arguments to the function are obtained. Since the value computed by f_v must be the same value as before, each new argument vertex must have a value identical to the old argument vertex value. We characterize this concept of vertex equality as follows.

Definition 4.4.1:

Let u and v be vertices in dependency graph D such that v is a transitive successor of u. Let dependency graph G be the projection of D on the vertices and edges that connect u to v. The vertex functions for G are as in D except each argument represented by an edge (w, x), where x is in G and w is not, is replaced by a constant equal to val_w. We call the function of val_v on argument val_u the *implicit dependency* of v on u. If this function is equal to the identity function, we say that there is an *implicit identity dependency* between u and v.

Given a dependency graph D, we construct a minimal dependency graph MIN D by first connecting all vertices u and v such that there is an implicit identity dependency between u and v, and then removing all vertices and edges whose sole purpose is to communicate values from u to v. The vertices removed are precisely those vertices that no longer have transitively dependent result vertices. We claim that propagation over this graph is optimal with respect to the assumptions of the previous section.

4.4.2 Implicit Identity Dependency Classes

Since identifying an arbitrary function as the identity function is in general undecidable, we cannot hope to locate all implicit identity dependencies. There are, however, some classes of implicit identity dependencies that can be easily recognized. Since these dependency classes occur frequently in dependency graphs generated by graph evaluable schemes, a substantial performance benefit can be realized by providing the direct communication of vertex values. We form the minimal dependency graph with respect to this class of implicit identity dependencies.

Definition 4.4.2:

Let D be a dependency graph and let C be a class of implicit identity frequently in incremental systems. We will then construct data structures representing the minimal dependency graph with respect to these implicit identity dependencies and will incrementally update these data structures to reflect incremental changes made in the original dependency graph. Since a solution for the minimal dependency graph is a solution to the original dependency graph, propagation over the minimal graph will allow us to reach an incremental solution more efficiently than had we performed propagation over the original dependency graph. Figure 4.5.2 illustrates this process.

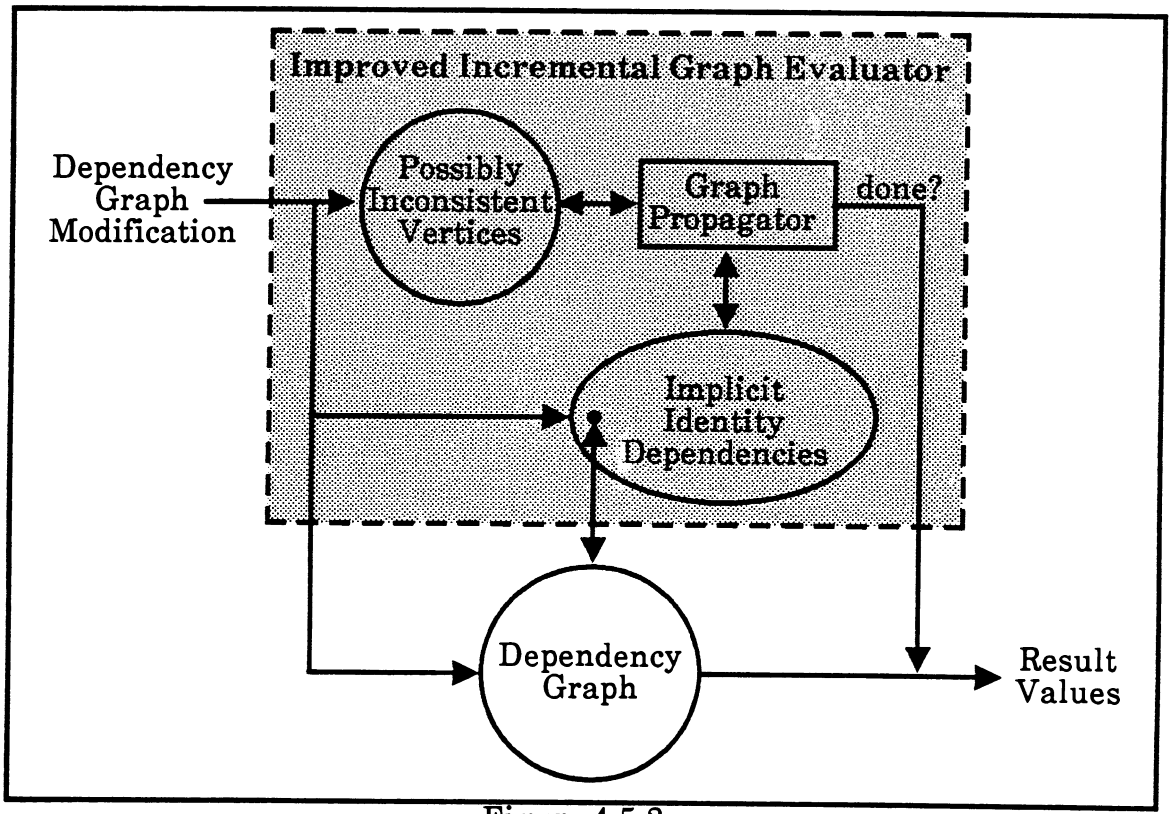


Figure 4.5.2

In Chapter 5, we develop the *structure tree* data structure. In the two chapters that follow, we use this data structure to maintain the implicit identity dependencies. We do this for the copy rule chain class of implicit identity dependencies in Chapter 6. In Chapter 7, we use structure trees to represent the implicit identity dependencies created by aggregates.

4.6 Relationship to Previous Work

The graph propagation algorithm of [Reps 84] was characterized as an optimal incremental evaluator for attribute grammars since it required O(|AFFECTED|)

5. Structure Trees

5.1 Introduction

In Chapters 6 and 7, we will show how two classes of implicit identity dependencies can be made explicit. The edges that make up the implicit identity dependencies comprise trees that are embedded in the dependency graph. The vertices of the trees represent intermediate computations that make up the implicit identity dependencies. A subset of the tree vertices correspond to vertices at the end of the implicit identity dependencies originating at the tree root. In order to bypass the implicit identity dependencies, we need to communicate the vertex value at the tree root to the vertex values of this vertex subset.

Figure 5.1.1 shows an example of a dependency tree. Vertices at the end of implicit identity dependencies are designated by squares. The goal of this chapter is to construct a data structure which will allow us to transmit the vertex value at the head of this tree to the square vertices in time proportional to the number of square vertices.

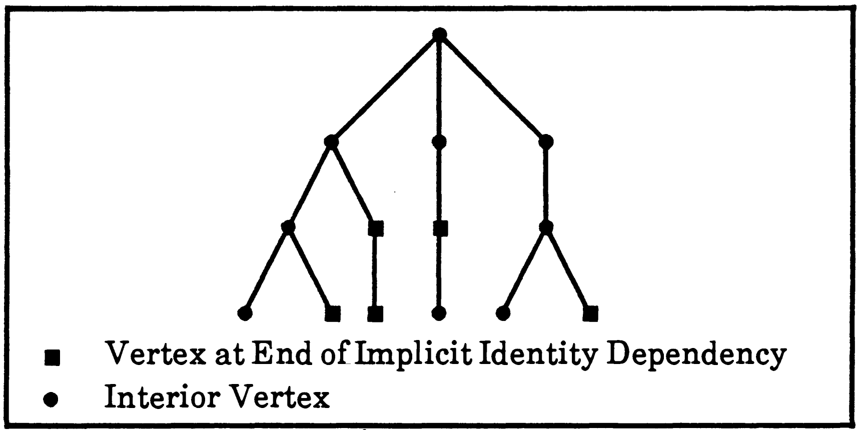


Figure 5.1.1

5.2 The Structure Tree

To facilitate communication between the root of the tree and the designated vertices, we create and incrementally maintain a structure preserving projection of the tree on the vertex subset. We call this projection a *structure tree*. In this chapter we discuss the construction and maintenance of a structure tree given a tree and a subset of its vertices. In the subsequent two chapters, we apply the algorithms of this chapter to explicitly maintain implicit identity dependencies.

Proof:

Let v_1 and v_2 be vertices in SV. Since the least common ancestor of v_1 and v_2 in T is also in SV, $(v_1, v_2) \in SE$ implies that v_1 is an ancestor of v_2 in T and that (v_1, v_2) is a tree edge.

Lemma 5.2.2:

Let T be a tree with vertex subset N. The size of the structure tree of T for N is $O(|N|)$.

Proof:

Let $T=(V, E)$ be an arbitrary tree and let $ST=(SV, SE)$ be the structure tree of T for N. Since $|ST|$ is $O(|SV|)$, the result follows from Lemma 5.2.1 and since $|SV| \leqq 2|N|$. We show the latter by induction on the size of N.

base: $N=\varnothing$

$N=\varnothing \Rightarrow SV=\varnothing \Rightarrow |SV| \leqq 2|N|$ by the structure tree definition.

inductive step: Assume $|SV| \leqq 2|N|$, let $ST'=(SV', SE')$ be the structure tree of T for $N \cup \{v\}$. We must show that $|SV'| \leqq 2(|N|+1)$.

If v is already in SV, $SV=SV'$ and $|SV'| \leqq 2(|N|+1)$. Otherwise, we add v to SV' with rule 1 of the structure tree definition. If $N=\varnothing$, we must also add the root of T to SV' for rule 3. If $N \neq \varnothing$, the root of T is already in SV. In this case we must argue that rule 2 adds at most one vertex. This implies that $|SV'| \leqq 2(|N|+1)$.

Assume that the second rule adds more than one vertex. Let $x_1 = LCA(v, y_1)$ and $x_2 = LCA(v, y_2)$ be two of these vertices. Since x_1 and x_2 are both ancestors of v, one must be an ancestor of the other. Assume x_2 is an ancestor of x_1. But then $x_2 = LCA(y_1, y_2)$ and must already be in SV. Thus we cannot add more than one vertex.

In the remainder of this chapter, we show how to maintain the structure tree of T in the presence of modifications to T. In Section 5.3, we introduce the s-tree data structure that is used to implement structure trees. As the s-trees will be representing implicit identity dependencies, they must be updated as dependency graph modification steps alter the dependency graph. In particular, dependency graph modifications may cause trivial vertices to become nontrivial (or vice versa), structure trees to become separated, and several structure trees to be combined into

The *s-tree_child_number* field contains the index of the structure tree node in the *s-tree_child* array of the node's parent. Alternatively, this information could be obtained with a search of the children of the parent. If the underlying tree has bounded outdegree, this would not affect the asymptotic running time of the structure tree algorithms.

The field *s-tree_parent* is a pointer to the parent node, and *s-tree_child* is an array of pointers to the children of the node in the structure tree. The number of array elements is the same as the outdegree of the corresponding vertex in the underlying tree. If there is no structure tree child in the subtree corresponding to a given edge in the underlying tree, the *s-tree_child* array element will be *nil*. Thus, structure tree children can be added in O(1) time.

5.4 Basic Routines for Structure Trees

In this section, we introduce the basic routines that will allow us to give concise algorithms in the next section for incrementally maintaining structure trees as the underlying tree changes.

We need to create s-tree nodes that correspond to vertices in T. This is done by the *new_s-tree_map* function shown in Algorithm 5.4.1. We assume that the function *new_s-tree* returns a new s-tree node with *nil* parent and child fields. *New_s-tree_map* installs this s-tree node at the given vertex and returns it.

```
new_s-tree_map(v : ↑ vertex) : ↑ s-tree
        v_s←new_s-tree();
        vertex_s-tree(v)←v_s;
        s-tree_vertex(v_s)←v;
        vertex_on_path(v)←true;
        return(v_s)
```

Algorithm 5.4.1

We also need to modify portions of s-trees. The procedure *cut_s-tree* removes from the structure tree the subtree rooted at the argument s-tree node. The *graft_s-tree* procedure connects a subtree rooted at c to the s-tree node p, child i. We will only call *graft_s-tree* when the appropriate s-tree child of p is *nil*. These procedures are shown in Algorithm 5.4.2.

v is no longer the root, if v is trivial and v_s has only one child, v_s should not be in the structure tree.

Procedure *path_compress*, shown in Algorithm 5.4.3, is used to take care of these inconsistencies. The function *only_s-tree_child* returns the child of the given s-tree node if there is only one. Otherwise, it returns *nil*. The function *any_s-tree_children* is true if and only if the structure tree node has children.

```
path_compress(v_s : ↑ s-tree)
        v←s-tree_vertex(v_s);
        oc_s←only_s-tree_child(v_s);
        p_s←s-tree_parent(v_s);
        if ¬is_nontrivial(v) then
                if p_s=nil and ¬any_s-tree_children(v_s) then
                        vertex_s-tree(v)←nil;
                        vertex_on_path(v)←false;
                        dispose(v_s)
                else if p_s≠ nil and oc_s≠ nil then
                        cut_s-tree(v_s);
                        cut_s-tree(oc_s);
                        graft_s-tree(p_s, oc_s, s-tree_child_number(v_s));
                        vertex_s-tree(v)←nil;
                        dispose(v_s)
```

Algorithm 5.4.3

In order to know which structure tree edges need to be changed, we must be able to follow paths in the underlying tree and locate vertices with structure tree nodes. We use the three procedures of Algorithm 5.4.4 for this purpose.

Find_mark locates the child of a vertex in the underlying tree that is on a structure tree path. There must be such a child if the vertex is on the path and does not have an s-tree map. It returns the index of the child.

Find_other_mark is similar to *find_mark*, but locates an on-path child whose index is not equal to the integer argument given.

Find_next_s-tree_map locates a vertex v that is the next vertex with a corresponding structure tree node on the path to the root. It also returns the index of v's child that is

```
mark_to_next_mark(v : ↑ vertex) : ↑ vertex×integer
        a←parent(v);
        a_i←child_number(v);
        while ¬vertex_on_path(a) and ¬is_root(a) do
                vertex_on_path(a)←true;
                a_i←child_number(a);
                a←parent(a);
        return(<a, a_i>);

unmark_to_next_s-tree_map(v : ↑ vertex) : ↑ vertex×integer
        a←parent(v);
        a_i←child_number(v);
        while vertex_s-tree(a)=nil and ¬is_root(a) do
                vertex_on_path(a)←false;
                a_i←child_number(a);
                a←parent(a);
        return(<a, a_i>)
```

Algorithm 5.4.5

The following theorem shows that we can build the structure tree within a constant space and time factor of the size of the underlying tree T.

Theorem 5.5.1:

Let T be a tree with n vertices. The s-tree for T can be built using $O(n)$ time and space.

Proof:

The *build_s-tree* procedure of Algorithm 5.5.1 builds the s-tree. Since it examines each vertex and edge of T no more than a constant number of times, it requires $O(n)$ time. Similarly, it uses only a constant amount of space for each vertex and edge of T.

5.6 Maintaining S-trees Incrementally

Since we will allow graph modification operations on our dependency graphs, we must be prepared to alter the structure trees to reflect modifications in the underlying dependencies. In the remainder of this chapter, we develop algorithms to incrementally maintain structure trees in the presence of changes to the underlying

When changes have been made, we would like to recompute the structure tree of T for N. As the underlying trees could be of arbitrary size, the cost of rebuilding the structure tree from scratch could be far greater than the size of the modification. In order to bring this cost closer to the size of the modification, we will incrementally maintain the structure tree. We will split the structure trees with the underlying tree so that each component has a valid structure tree, and when we combine two underlying trees together, we will join the structure trees of the components so that we have the structure tree of the resulting underlying tree.

5.6.1 Changing a Vertex to be Nontrivial

If a vertex status changes from trivial to nontrivial, we add the vertex v using the *bind_vertex* procedure of Algorithm 5.6.1. There are two major cases. Either v is on a structure tree path or it is not.

If v is on the structure tree path and already has a structure tree node associated with it, we are done. Otherwise, we have the case shown at the top of Figure 5.6.1. We climb the underlying tree until we come to the first vertex p with an associated structure tree node. We cut the structure tree edge to the child c of p that is below v and we graft the appropriate edges so that p has child v which in turn has child c.

If v is not on the structure tree path, we find the first vertex on the path to the root that is on a structure tree path. If there is no such vertex, the structure tree is empty and we must create a structure tree node at the root. Otherwise, as shown at the bottom of Figure 5.6.1, we bind this on-path vertex. In either case, we have the structure tree parent of v and we link it to a new structure node corresponding to v.

The total time required by this operation is O(*path* + *outdegree*) where *path* is the path length in the underlying structure from the changed vertex to its first ancestor in the structure tree, and *outdegree* is the outdegree of the least common ancestor vertex. If *outdegree* is bounded by a constant, this is O(*path*). Otherwise, the index of vertex's successor that is on the path could be stored in each vertex to give O(*path*) time.

5.6.2 Removing Edges of the Underlying Tree

When we remove an edge in T, we must determine if removing that edge will affect the structure tree, and if it does, we must modify the structure tree appropriately. If the edge removal does not affect the structure tree, we need to do nothing other than the normal tree surgery on T.

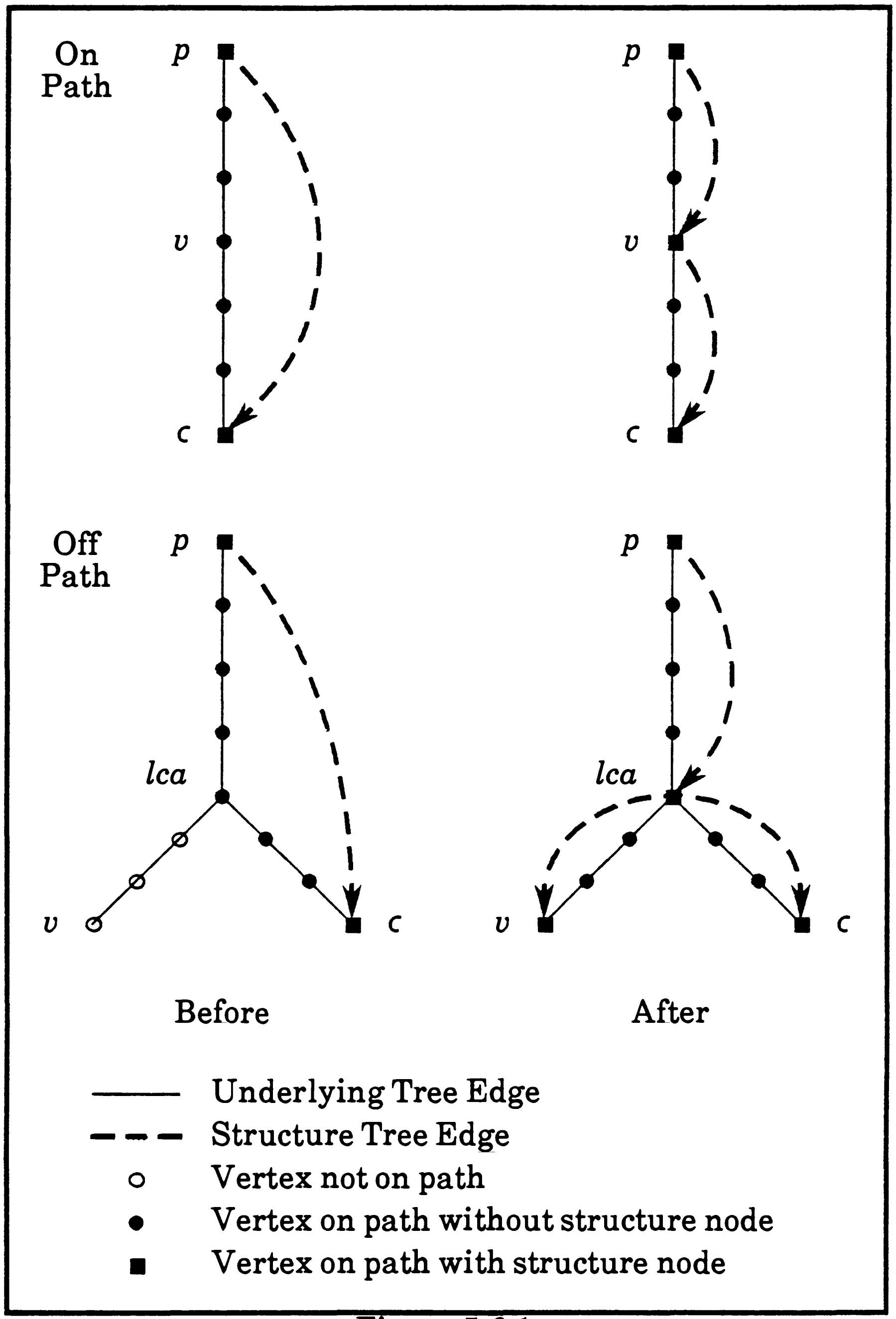


Figure 5.6.1

corresponding structure tree node. Cutting off the correct child of this node and calling *path_compress* (Algorithm 5.4.3) restores the upper structure to consistency.

We restore the lower tree as follows. If v has a corresponding structure tree node, it is a valid structure tree. Otherwise, there exists a nontrivial vertex somewhere below v

unmark the path from v up to the next underlying vertex with a structure tree map and remove the structure tree edge. This is done by calling *split_s-tree* on v, and then calling *path_compress* on v to remove the trivial vertex. The total time required is O(*path*).

The copy rule tree represents a set of implicit identity dependencies from the copied vertex. If the value at this vertex changes, incremental propagation requires that all of the vertices in the copy rule tree be updated. Since the copy rule chains could be of arbitrary length, the performance of an incremental evaluator would be improved if we could communicate the value directly from the copied vertex to all vertices dependent upon the tail of a copy rule chain. It is the goal of this chapter to show how these values can be communicated during incremental evaluation in time independent of the length of these chains.

6.2 Why Copy Rule Chains Occur

There are several reasons why copy rule chains occur in dependency graphs constructed by graph evaluable schemes. First, in some graph evaluable schemes, communication can only take place between vertices local to some structure external to the dependency graph. For example, in attribute grammars, the dependency graph is embedded in an abstract syntax tree. Communication is limited to vertices in the same production of the context free grammar used to specify the abstract syntax tree.

Second, copy rules allow a given value to be broadcast throughout the dependency graph. For bounded graph evaluable schemes, copy rules become necessary as the number of vertices that a given vertex could communicate its value to would otherwise be bounded.

6.3 Bypassing Copy Rule Chains

We bypass copy rule chains by incrementally maintaining the structure tree of each copy rule tree in the dependency graph D.

Definition 6.3.1:

> Let T be a copy rule tree in D that is rooted at v and let N be the set of all copy vertices in T that have direct successors that are outside of T. We refer to each element of N as a *nontrivial copy vertex* and call the structure tree of T for N the *copy structure tree.*

It is the direct successors of the nontrivial copy vertices that are implicitly dependent upon the value of the copied vertex v. We extend the notion of a consistent dependency graph D to include the existence of the copy structure tree for each copy rule tree in D. Section 6.4 discusses how the structure trees are built during the initial (non-incremental) graph evaluation. In Section 6.5, we show how we can translate graph

For some graph evaluators we can gain a constant factor of improvement as follows. Since all copy vertices in a copy rule tree T are dependent upon v, the root of T, they must follow v in all topological orders of D. Likewise, since all of these copy vertices have indegree 1, a preorder traversal of these vertices after visiting v will correspond to some topological order of D.

This fact allows us to alternate between a topological sort and the *build_s-tree* procedure while performing topological evaluation. The topological sort begins and evaluates the vertex functions associated with each vertex until a vertex v is reached with a direct successor whose value is defined by a copy rule. We evaluate v and attribute the vertex with the resulting value. We then call *build_s-tree* (Algorithm 5.5.1) on v with the following modifications.

First, we need to attribute the vertices with the appropriate values. Since all of the vertices in the copy rule tree have the same value, we pass this value through all recursive calls to *build_s-tree* and assign this value to all vertices with s-tree maps. Second, we need to be able to continue the topological sort after the *build_s-tree* procedure has finished. This requires that we decrement the indegree of noncopy successors of nontrivial vertices, and add them to the worklist if the indegree goes to zero.

6.5 Dependency Graph Modifications

We allow dependency graph modifications as described in Section 2.3. These modifications can alter the copy rule trees, which in turn correspond to changes in the copy structure tree. In order to maintain the consistency of the copy structure trees, we must perform the appropriate structure tree modification each time a modification step is performed on the dependency graph.

The modification steps that make up a graph modification are *change function, delete component, add component, add edge*, and *remove edge*. In this section, we show how the structure tree operations necessary for these modification steps can be performed with the structure tree operations of Section 5.6. We will augment each graph modification step with these structure tree operations.

Since changing a vertex function does not affect the edges of the dependency graph, the *change function* does not require any structure tree operations. The *delete component* operation does not require any structure tree changes either, since the

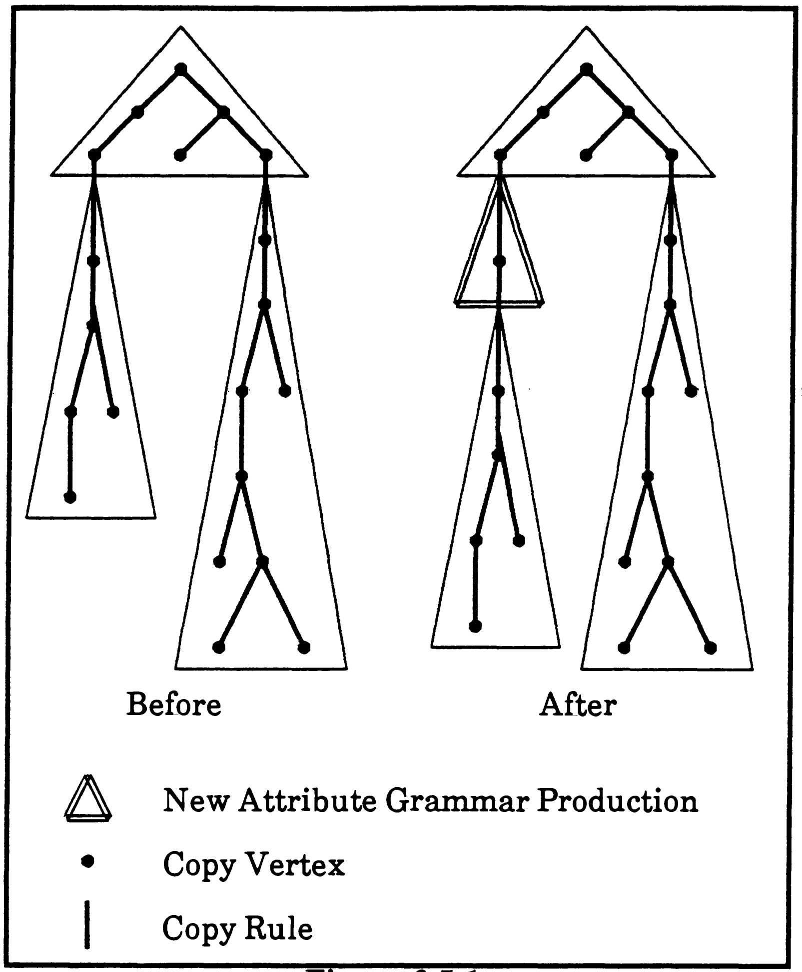


Figure 6.5.1

6.6 Graph Propagation with Structure Trees

We have specified how we keep the copy structure trees consistent with their corresponding copy rule trees. We need to show how these trees can be used to allow faster incremental evaluation.

Note that the the root of a given structure tree is accessible from the root of the corresponding copy rule tree, and that the corresponding vertex in D can be reached from each vertex in the structure tree. This allows us to view D as a different dependency graph D' that has all of the copy rule trees replaced by the corresponding copy structure trees. The root of the copy structure tree retains the vertex function of the copy rule tree had. The vertex function assigned to all other vertices of the copy structure tree, is the identity function. Since the copy vertices that are omitted from the copy structure tree are not relevant to the computation, the solution of D' is the copy structure trees. Let C be the class of implicit identity functions induced by copy rule chains. Graph propagation over D' can be done in time that is within a constant factor of the time required for graph propagation over $MIN_C D$.

Proof:

$MIN_C D$ is identical to D' except that D' contains structure trees whose nontrivial vertices are connected to vertices that are implicitly dependent upon the root of the corresponding copy rule tree. Let v be the root of this tree. In $MIN_C D$, the implicitly dependent vertices are directly connected to v. Theorem 5.2.1 implies that we can traverse each structure tree in D in time proportional to the number of nontrivial vertices in that structure tree, m. Since $MIN_C D$ has at least one edge for each of these nontrivial vertices, and propagation must be done over the edges, we perform $O(m)$ propagation steps in both cases.

The above theorem, however, does not claim optimality for the maintenance of the data structures necessary to facilitate this propagation. In Section 6.7, we compare this cost with that of other methods.

6.7 Relationship to Previous Work

We note the similarity between the copy structure tree and the *copy bypass tree* of [Hoover 86]. Both trees contain the nontrivial copy vertices. The copy bypass tree is a balanced tree that contains exactly the nontrivial copy vertices while the copy structure tree is not balanced and contains other copy vertices to retain the same structure as the copy rule tree.

Both structures allow the nontrivial copy vertices, N, to be found in time proportional to $|N|$. The major performance difference between the two methods is the cost required to store the structures and the time required to update the structures after a dependency graph modification that affects copy rule trees.

When a copy rule tree is split or joined the copy bypass method requires, in general, that the entire copy subtree of the modification be traversed, even if these vertices are outside of INFLUENCED. In addition, a balanced tree split must be performed on the copy bypass tree requiring $O(\log |N|)$ path comparisons, where each path can have length O(copy rule tree height). The method also requires that the paths be stored in the copy bypass tree.

graph. Our techniques can be extended to allow multiple operations on a single aggregate at a given vertex if we consider the set of operations to be a single operation on a set of keys.

To extend aggregate values to be first class objects, one must move the information used to represent implicit identity dependencies from the vertices into the vertex values. Additional research is needed to determine if this approach is practical.

Definition 7.1.2:

> An aggregate vertex whose vertex function is ADD is called a *definition site*. An aggregate vertex whose vertex function is LOOKUP is referred to as a *use site*. We call the set of definition sites for a given aggregate *DEFINITIONS*, and the set of use sites, *USES*.

If we assume that the elements being placed in the aggregate are themselves vertex values, the aggregate represents implicit identity dependencies between these vertices and the use sites. In this chapter, we extend the methods of the previous chapter to allow these implicit identity dependencies to be represented by structure trees.

To see why this optimization is important, one must examine the structure of an aggregate in a dependency graph. Figure 7.1.2 shows how elements might be put into and taken out of aggregate values. The letters by the definition and use sites indicate the key used for that element.

Consider the effect of incremental propagation after a change in the value of the element inserted with key a. Graph propagation will require us to do bookkeeping work proportional |INFLUENCED|, in addition to |INFLUENCED| vertex function evaluations. In this example, AFFECTED will include the 11 dependent aggregate valued vertices (all copy and definition sites) plus both use sites of a. The INFLUENCED set will include AFFECTED plus all of the use sites.

In this structure, a change in a definition for key k only has the potential to affect the uses of that particular key. Assuming that the aggregate values affected by the change are only used to lookup and define keyed values, their specific values are not directly relevant to the aggregate computation. Instead, it is the identity dependencies implicit in this structure that are relevant.

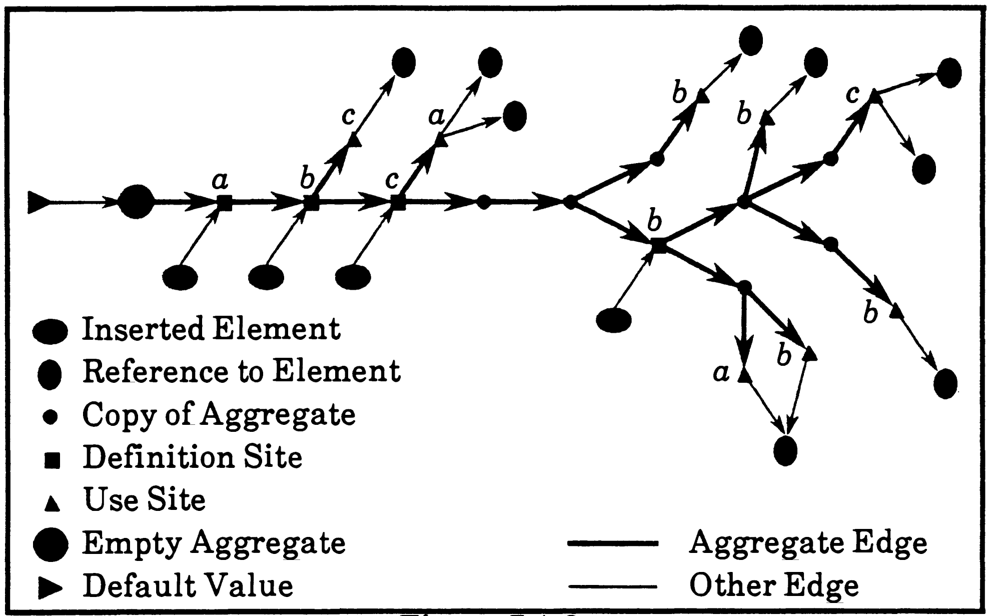


Figure 7.1.2

Let USES/k be the set of uses that refer to the key k. If we can make the implicit identity dependencies between uses and their definitions explicit in the dependency graph, we can reduce the amount of computation required for updating the aggregate after a change in the definition for k from O(|DEFINITIONS| + |USES|) closer to O(|USES/k|). Since these aggregate values can become quite large, and since operations upon them are typically expensive, this would be a substantial performance improvement in incremental evaluation.

In the the rest of this chapter, we discuss efficient implementation strategies for collections, which are aggregates that can be constructed with the aggregate operations of Figure 7.1.1. Collections will allow us to determine and maintain the identity dependencies implicitly specified between the definition and use sites of an aggregate. We first investigate static collections, which give ADD and LOOKUP functionality with constant keys. We then give extensions for dynamic collections, which allow keys to be computed, and for nested collections, which allow the UPDATE operator to be used.

The remainder of this chapter is organized as follows. In Section 7.2, we discuss how structure trees can be used to represent the implicit identity dependencies created by collections. The aggregate operators form a tree, called the collection tree, imbedded in the dependency graph. A structure tree of the *collection tree*, called the *base tree*, is used to bypass copy operators in the collection. The communication of each definition to its dependent uses is represented by a structure tree of the base tree called a key tree. Therefore, each aggregate will correspond to numerous structure trees. Figure 7.1.3 shows these trees for the example in Figure 7.1.2. We will continue with this example in Sections 7.2, 7.3, and 7.4.

Section 7.3 extends the structure tree algorithms of Chapter 5 so that structure trees representing static collections (collections where the keys are statically defined) can be built and incrementally maintained. Since there can potentially be a key tree for each key at a given vertex of the aggregate, the structure tree algorithms must be extended to update a set of structure trees.

Propagation using the structure trees is discussed in Section 7.4. An improved dependency graph is constructed by replacing the aggregate dependencies with the dependencies represented by the key trees, allowing efficient propagation. We consider several data structures for representing the key tree sets in Section 7.5.

The static collection techniques of Section 7.3 are extended for dynamic collections, which allow keys to be computed by the aggregate, in Section 7.6, and for nested collections, which allow collections to be combined with the UPDATE operator, in Section 7.7. We compare the results of this chapter with relevant previous work in Section 7.8.

7.2 Collections

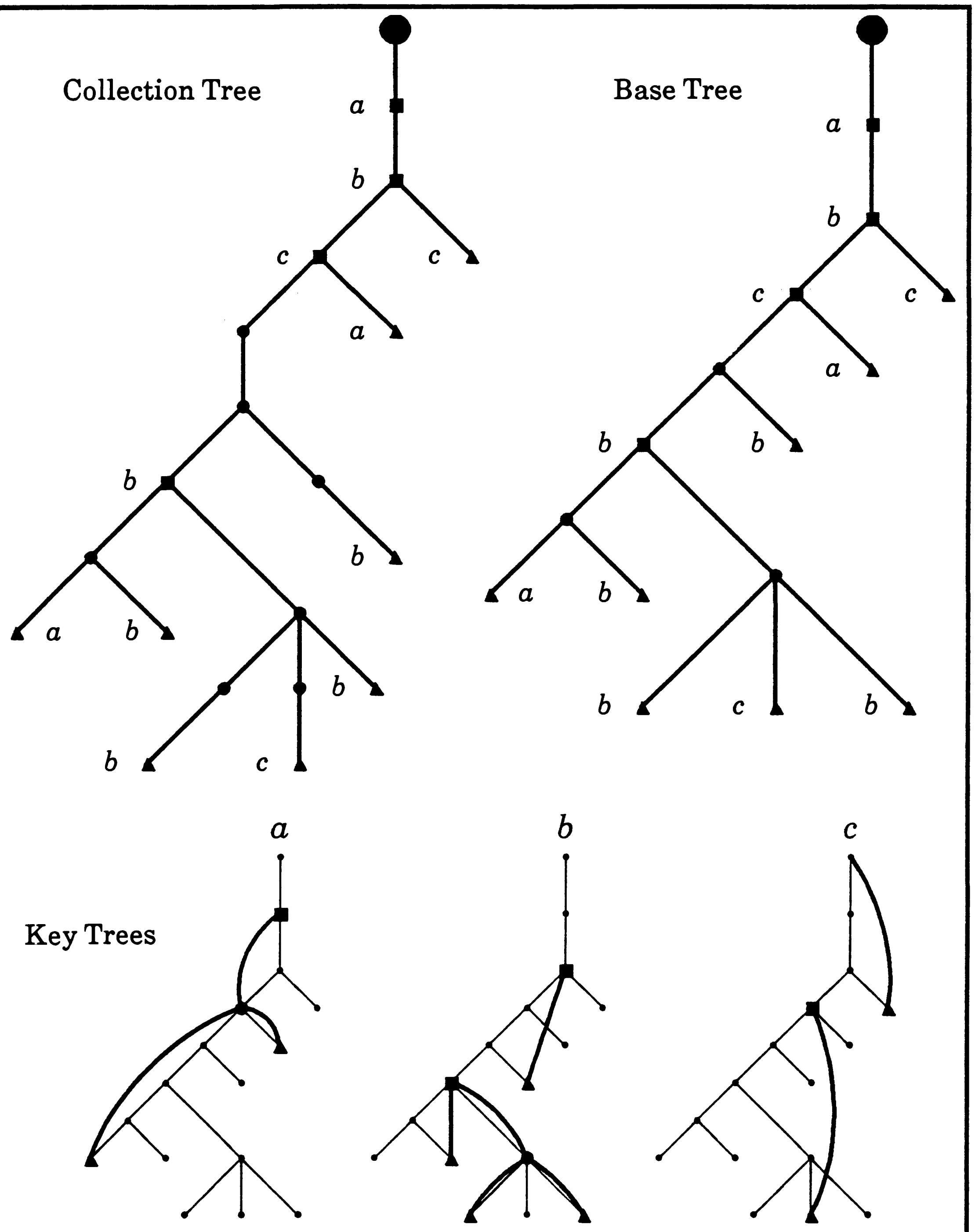
In this section we consider aggregates that can be formed using the EMPTY, ADD, LOOKUP, and COPY operators.

Definition 7.2.1:

A *collection* is an aggregate that is constructed in a dependency graph using the EMPTY, ADD, LOOKUP, and or COPY operators. The maximal tree that connects the corresponding aggregate vertices over aggregate edges is called a *collection tree.* If the keys specified in the ADD and LOOKUP operators are constants given in the vertex functions for these vertices, the collection is called *static.* Otherwise, the keys may be computed by the dependency graph and the collection is *dynamic.*

Observation 7.2.1:

The collection tree is indeed a tree and is rooted at an aggregate vertex

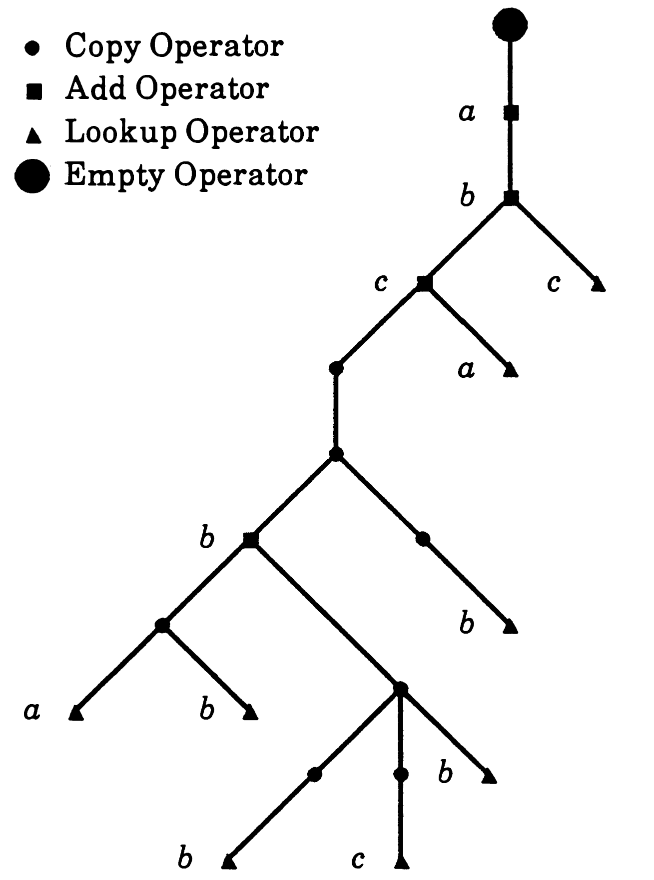


Trees Used to Represent Communication in Aggregates

Figure 7.1.3

corresponding to the one and only one EMPTY aggregate operator in the collection.

Figure 7.2.1 shows the collection tree that corresponds to the aggregate example of Figure 7.1.1.



Collection Tree for Figure 7.1.1

Figure 7.2.1

We locate and maintain the implicit identity dependencies in collections using structure trees as we did for copy rules in the previous chapter. Both form tree structures nested among the edges of the dependency graph. The primary difference is that in the case of copy trees, only one value was communicated over each edge of the copy tree, whereas with collections, we have m values being communicated, where m is the size of the aggregate at the given collection tree edge.

7.2.1 The Base Tree

As in the case of copy rules, the collection tree will have trivial vertices introduced by the COPY operator. We bypass these vertices by forming the structure tree of the collection tree with the ADD and LOOKUP vertices as the nontrivial vertex set. We call this tree the *base tree* of the collection—it represents the transmission of the aggregate value in the dependency graph.

The base tree is to its collection tree as the copy structure tree is to the copy rule tree in Chapter 6. Theorem 6.4.1 applies to the base tree as well, implying that we can construct the base tree with a constant factor of time and space overhead.

Figure 7.2.2 shows the base tree for the collection tree example of Figure 7.1.1. In this figure, each base tree edge is labeled with a tuple representing the path in the collection tree. We number the edges of a given vertex from 0 to the number of edges -1. For example, the path $0, 1$ represents the path from the corresponding vertex in the collection tree through the 0^{th} child and to the 1^{st} child of the 0^{th} child.

7.2.2 Key Trees

In the case of copy rules, we represented the communication of a single value through a tree of dependencies with a structure tree. In the case of collections, an edge in the collection tree corresponds, in general, to the communication of a number of values. At any given edge, however, at most one value can be bound to a given key k.

Since the value defined for a given key k can be redefined at a different point in the collection tree, we cannot use just one structure tree to represent the communication of the value for k. However, since there can only be one value for k at any given edge or vertex, the structure trees used for k cannot overlap. Therefore, we can partition the base tree for each key k and create a structure tree to represent the communication of the value for k in each partition. We call these structure trees *key trees*. Each key will be associated with a forest of such trees embedded in the base tree.

For key k, we partition the base tree into smaller trees within which k is bound to unique values. At the root of the base tree, key k is bound to the default value, which is given by the argument of the EMPTY operator. New values are bound to k by ADD operations for key k.

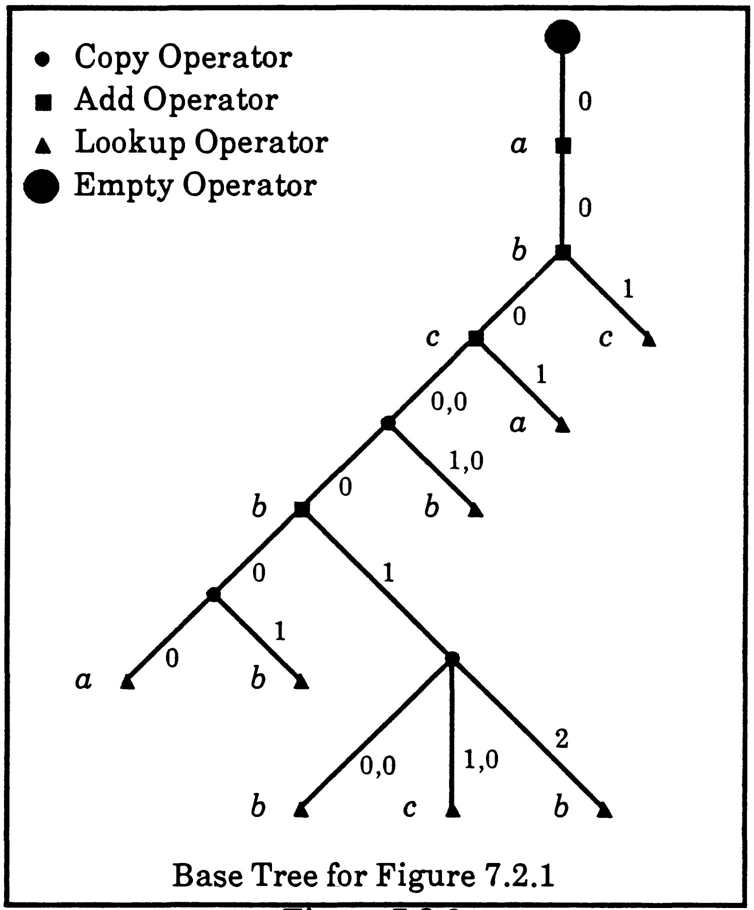


Figure 7.2.2

Definition 7.2.2:

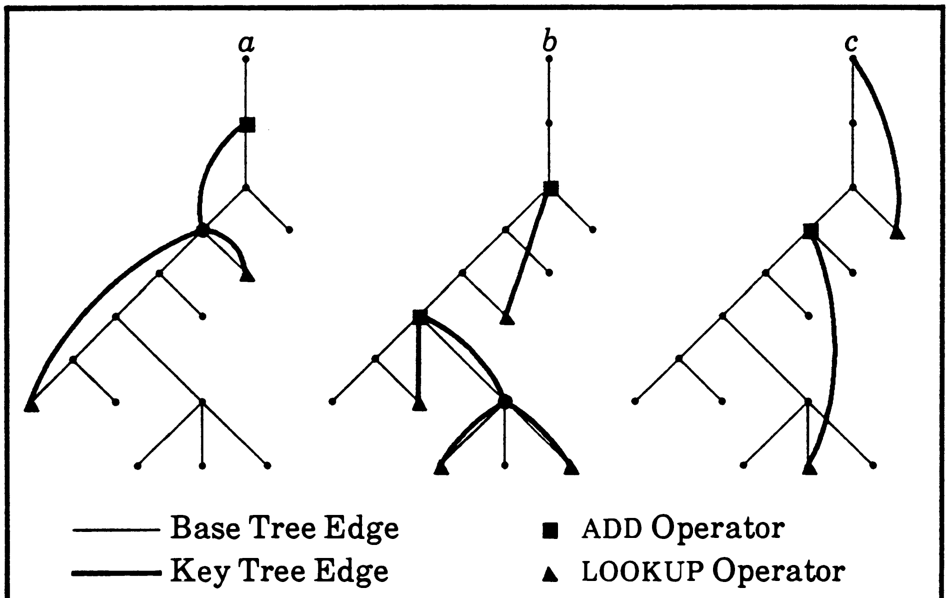
Let ROOT(k) be the set of vertices that contains all ADD vertices in the base tree for key k as well as the root of the base tree. The forest of nonintersecting trees that are rooted at vertices in ROOT(k) and contain all descendants of the base tree down to, but not including, the root of the next such tree is called the *base forest* for k.

Each of the trees in the base forest for k will have a structure tree to represent the communication of the value from where it is defined, to the LOOKUP vertices where the value is referenced.

Definition 7.2.3:

For each tree T in the base forest for k, let N_T be the LOOKUP vertices for key k in T, and let K_T be the structure tree of T for N_T. We call each K_T a *key tree*, and refer to the forest of these trees as the *key forest* for k.

According to the definition of a structure tree (Definition 5.2.1), the nodes in each key tree correspond to the use sites in the base tree where the value for *k* is needed, the least common ancestors of these nodes in the base tree, and the definition point of the value for key *k*, if the defined value is in fact needed. The key trees corresponding to the base tree example of Figure 7.2.2 are shown in Figure 7.2.3.



Key Trees for Figure 7.2.2

Figure 7.2.3

Using the data structure of Chapter 5, we have pointers that map from vertices in the dependency graph to nodes of the base tree and vice versa. Since a given base tree node can correspond with nodes of each of several key trees, we will extend the *vertex_s-tree* field to be a set of pointers to these key tree nodes. These pointers are indexed by the appropriate key. Each key tree node has a pointer back to the base tree node. We extend the boolean *vertex_on_path* field to be a set of keys, indicating that the base tree vertex is on a path in a key tree for each of these keys.

The key tree nodes can be stored with a constant factor of space overhead as the following theorem shows.

Theorem 7.2.1:

The number of nodes required by the key trees is O(|V|) where V is the set of collection tree vertices.

Proof:

Follows from Lemma 5.2.2 since each vertex in V can only be nontrivial with respect to a single key.

Unfortunately, we cannot place a similar bound on the size of the *vertex_on_path* sets. Let K be the set of keys used in the aggregate. A straightforward, unshared implementation of these sets could require up to O(|K|) space at each vertex. We will return to this problem in Section 7.5, where we discuss methods of representing the *vertex_on_path* and *vertex_s-tree* sets.

In the remainder of this chapter, we will show how collections can be implemented and used for efficient propagation. In Section 7.3, we extend the structure tree algorithms of Chapter 5 to allow us to maintain data structures representing the implicit identity dependencies in static collections. We use these data structures to speed propagation in Section 7.4. Section 7.5 discusses methods of storing the *vertex_s-tree* and *vertex_on_path* sets. We give extended techniques for dynamic collections in Section 7.6. In Section 7.7 we extend collections to allow use of the UPDATE operator.

7.3 Static Collections

In this section, we discuss algorithms for maintaining the base and key tree data structures for static collections.

7.3.1 Building and Maintaining the Base Tree

In addition to the s-tree data structure fields described in Section 5.2, the base tree has several other fields. The type definitions for these fields are shown in Figure 7.3.1.

```
base-tree : record of
        ... s-tree fields of Figure 5.3.1 ...
        vertex_key : key;
        vertex_s-tree : set of ↑ s-tree;
        vertex_on_path : set of key
```

Figure 7.3.1

The s-tree fields are used to maintain the base tree exactly as was done in Chapter 5. The *vertex_key* field is used to store the key of the key tree to which the corresponding aggregate vertex is associated. This is only meaningful for vertices defined by the ADD or LOOKUP operators. We assume that this field is initialized to *nil*.

The *vertex_s-tree* field is the set of key tree vertices that correspond to this base tree vertex. The *vertex_on_path* set contains the keys of all key trees that have edges that pass through or over the base tree vertex. These sets are the analog of the dependency graph vertex fields of the same name, but are extended for multiple structure trees as described in Section 7.2.2. We assume that both of these fields are initialized to be the empty set.

We use the same algorithm to build the base tree that we used to build the copy structure trees in Section 6.4. In this setting, however, the root of the collection tree is recognized by having a vertex value defined by the EMPTY operator. Likewise, the underlying tree is the connected region of vertices defined by the aggregate operators, all of which are nontrivial except for those defined by the COPY operator. The result is the construction of a base tree for each collection tree in the dependency graph.

We maintain the consistency of these base trees as we did for copy structure trees in Section 6.5. This includes the addition and removal of edges in the underlying tree as well as the initialization of new components.

7.3.2 Building the Key Trees

Since the key trees for a static collection are totally determined by the dependencies and vertex functions of the collection, we can build the key trees in a single pass over the base tree. In the same manner as the *build_s-tree* procedure (Algorithm 5.5.1), we can perform a postorder traversal of the base tree, building structure trees bottom up. In *build_s-tree*, however, we had only one structure tree. Instead of returning from each recursive call with a single structure subtree, we extend this procedure to build multiple structure trees by returning a set of structure subtrees, one for each undefined key used in the corresponding base subtree.

The procedure for constructing the key trees, *build_key_trees*, is shown in Algorithm 7.3.1. At each vertex, the key subtrees for each child are computed. At the root of the base tree, we create structure tree nodes for all unbound keys and connect them to the subtrees below. At other vertices, we create a root for the key of an ADD vertex, and create least common ancestor nodes for the keys that appear in more than one subtree. The *vertex_on_path* set for the vertex is all keys in the subtrees plus the key used at the vertex (for ADD and LOOKUP vertices). *Vertex_s-tree* is the set of s-tree nodes created at the vertex.

```
build_key_trees(v: ↑ vertex) : set of (key × ↑ s-tree)
        for i←0 to #children − 1 do
                S[i]←build_key_trees(child(v, i));
        T←∅;
        if v is defined by EMPTY operator then
                merge S[0], S[1], ..., S[#children − 1]. For each key k that
                appears in any set, create a new s-tree node n for k. Connect
                each node for k in S[i] as the ith child of n, and then add <k, n>
                to T
        else if v is defined by ADD operator for key l then
                vertex_key(v)←l;
                merge S[0], S[1], ..., S[#children − 1]. For each key k≠l that
                appears in a single set S[i], add this set element to T.
                Otherwise, for k=l that appears in any S[i] and for each key
                k≠l that appears in more than one set, create a new s-tree node
                n for k. Connect each node for k in any S[i] as the ith child of n.
                Add <k, n> to T if k≠l
        else if v is defined by LOOKUP operator for key l then
                vertex_key(v)←l;
                Create a new s-tree node n for l;
                T←{<l, n>}
        else
                merge S[0], S[1], ..., S[#children − 1]. For each key k that
                appears in a single set S[i], add this set element to T.
                Otherwise, for any key k that appears in more than one set,
                create a new s-tree node n for k. Connect each node for k in any
                S[i] as the ith child of n, and then add <k, n> to T;
        vertex_on_path(v)←{keys in set T};
        vertex_s-tree(v)←{new s-tree nodes created in T};
        return(T)
```

Algorithm 7.3.1

If we maintain the sets of key subtrees as ordered lists, the amount of computation required by the merge at each base tree vertex v is O(#children**key subtrees*$_v$), where *key subtrees*$_v$ is the number of key subtrees that correspond to the base tree subtree below v. Since the *vertex_on_path* set is O(*key subtrees*$_v$) in size, it can be constructed in O(*key subtrees*$_v$) time. As discussed in Theorem 7.2.1, the number of key tree nodes is O(|V|), where V is the set of base tree nodes This implies that the s-tree nodes and the *vertex_s-tree* sets can be built at an amortized O(1) cost per base tree vertex.

It is possible to reduce the merge time required using the merging techniques of [Brown and Tarjan 79]. The persistent data structure methods of [Driscoll, Sarnak, Sleator, and Tarjan 86] could then be applied to allow the elements of the on-path sets to be shared, reducing both the storage and construction time. This topic is revisited in Section 7.5, where we discuss storage representations for the *vertex_on_path* sets.

7.3.3 Maintaining the Key Trees

While incrementally maintaining key trees in the presence of modifications to the underlying static collection tree is similar to the maintenance of the base tree, we must extend the algorithms of Chapter 5 to accommodate multiple structure trees. Although most of the basic operations on structure trees are similar, there are several differences.

- Due to the possibility of multiple definitions for keys, the underlying structure of each key tree may be a subtree of the base tree. The type *vertex* refers to a base tree vertex.
- The root of the underlying structure is the first ADD vertex for key k encountered on the path to the root of the base tree, or the base tree root if there are no ADD vertices for k. The predicate *is_root* depends, therefore, on k as well as v.
- A base tree vertex is nontrivial for key k if its vertex function is defined by LOOKUP, with k for the key argument.
- We must perform set operations on the *vertex_s-tree* and *vertex_on_path* fields of the base tree vertices.
- Most structure tree operations now require, as an additional argument, the key of the structure tree in question.

In the remainder of this section, we extend the algorithms of Chapter 5 to allow manipulation of the key trees. In the following discussion, we will use the variables shown in Figure 7.3.2 to characterize the cost of the key tree manipulation operations. We relate these variables to more concrete values in Section 7.5, where we discuss tradeoffs among the various data structures that could be used to represent the *vertex_s-tree* and *vertex_on_path* sets.

Variable	Meaning
outdegree	outdegree of vertex
path	length of the path rootward to next base tree vertex s-tree node for the same key
pmod	time for single element modification to *vertex_on_path* set
psize	size of *vertex_on_path* set
ptest	time for membership test on *vertex_on_path* set
ptrav	time required to traverse *vertex_on_path* set
smod	time for single element modification to *vertex_s-tree* set
ssize	size of *vertex_s-tree* set
stest	time for membership test on *vertex_s-tree* set
strav	time required to traverse *vertex_s-tree* set

Figure 7.3.2

7.3.3.1 Basic Routines for Key Trees

Since the *vertex_s-tree* field is now a set of key tree nodes, creating a new s-tree map now involves a set operation to add the new s-tree vertex. In addition, we must add the key to the *vertex_on_path* set. This *new_s-tree* function, which generalizes Algorithm 5.4.1, is shown in Algorithm 7.3.2. Total time required is O(*pmod* + *smod*).

```
new_s-tree_map(v : ↑ vertex, k : key) : ↑ s-tree
        v_s←new_s-tree();
        vertex_s-tree(v)←vertex_s-tree(v)∪{v_s};
        s-tree_vertex(v_s)←v;
        vertex_on_path(v)←vertex_on_path(v)∪{k};
        return(v_s)
```

Algorithm 7.3.2

The *path_compress* procedure of Algorithm 5.4.3 corrected structure tree violations that result from the cutting and grafting of structure tree edges. The extended *path_compress* operation, shown in Algorithm 7.4.3, compresses key tree k at the given vertex. The key is necessary to determine if the vertex is trivial. (A vertex v is trivial for key k if and only if it is a bound LOOKUP vertex for key k). Since we must remove the s-tree node, and possibly the on_path entry for violation #1 (see Section 5.4), this operation requires $O(pmod + smod)$ time.

```
path_compress(v_s : ↑ s-tree, k : key)
    v←s-tree_vertex(v_s);
    oc_s←only_s-tree_child(v_s);
    p_s←s-tree_parent(v_s);
    if ¬is_nontrivial(v, k) then
        if p_s = nil and ¬any_s-tree_children(v_s) then
            vertex_s-tree(v)←vertex_s-tree(v) − {v_s};
            vertex_on_path(v)←vertex_on_path(v) − {k};
            dispose(v_s)
        else if p_s ≠ nil and oc_s ≠ nil then
            cut_s-tree(v_s);
            cut_s-tree(oc_s);
            graft_s-tree(p_s, oc_s, s-tree_child_number(v_s));
            vertex_s-tree(v)←vertex_s-tree(v) − {v_s};
            dispose(v_s)
```

Algorithm 7.3.3

Likewise, the path searching and marking functions of Algorithms 5.4.4 and 5.4.5 are extended to search for s-tree mappings of the appropriate key. The running times of these extended operations are shown in Figure 7.3.3. The functions appear in Algorithms 7.3.4 and 7.3.5. The *lookup* procedure is used to locate the s-tree node in the *vertex_s-tree* set, for a given key.

Given these basic routines, we now extend the structure tree modification algorithms of Chapter 5 to allow modifications on key trees. In order to support incremental graph evaluation, we must support the graph modification operations of Section 2.3. In the rest of this section, we give the relevant key tree modification algorithms, and

Function	Time Required
find_mark	O(*ptest*outdegree*)
find_other_mark	O(*ptest*outdegree*)
find_next_s-tree_map	O(*stest*path*)
mark_to_next_mark	O((*ptest*+*pmod*)**path*)
unmark_to_next_s-tree_map	O((*stest*+*pmod*)**path*)

Figure 7.3.3

```
find_mark(v : ↑ vertex, k : key) : integer
        for i←0 to #children(v) − 1 do
                if k∈vertex_on_path(child(v, i)) then
                        return(i);

find_other_mark(v : ↑ vertex, m : integer, k : key) : integer
        for i←0 to #children(v) − 1 do
                if m≠i and k∈vertex_on_path(child(v, i)) then
                        return(i);

find_next_s-tree_map(v : ↑ vertex, k : key) : ↑ vertex×integer
        a←parent(v);
        a_i←child_number(v);
        while lookup(vertex_s-tree(a), k) = nil and ¬ is_root(a, k) do
                a_i←child_number(a);
                a←parent(a);
        return(<a, a_i>)
```

Algorithm 7.3.4

discuss how they are used to implement the graph modification steps. These steps are *delete component, add component, change function, add edge*, and *remove edge.*

7.3.3.2 Deleting or Adding a Component

When deleting a component, we simply remove all of the key trees that correspond to any base trees removed. Upon the addition of a new component, the added vertices are traversed as in Section 6.5 to construct any base trees in the new component. When the base tree has been constructed, we call *build_key_trees* (Algorithm 7.3.1) on the root of the base tree. This will construct the key trees for each collection in the

```
mark_to_next_mark(v : ↑ vertex, k : key) : ↑ vertex×integer
        a←parent(v);
        a_i←child_number(v);
        while k∉vertex_on_path(a) and ¬ is_root(a, k) do
                vertex_on_path(a)←vertex_on_path(a)∪{k};
                a_i←child_number(a);
                a←parent(a);
        return(<a, a_i>);

unmark_to_next_s-tree_map(v : ↑ vertex, k : key) : ↑ vertex×integer
        a←parent(v);
        a_i←child_number(v);
        while lookup(vertex_s-tree(a), k) = nil and ¬ is_root(a, k) do
                vertex_on_path(a)←vertex_on_path(a) − {k};
                a_i←child_number(a);
                a←parent(a);
        return(<a, a_i>)
```

Algorithm 7.3.5

new component. Later, when edges are added to connect these collection trees to other collection trees, we will join the appropriate key trees together. This is described in Section 7.3.3.4 below.

7.3.3.3 Changing a Vertex Function

The key forest for k is determined by the ADD and LOOKUP vertices for k in the collection tree. When a change in the vertex function of a vertex v changes the vertex *from* an ADD or LOOKUP vertex, we must remove the corresponding key tree nodes from the key forest and adjust the key trees accordingly. In addition, when a *change function* operation turns v *into* an ADD or LOOKUP vertex for key k', we must correct the key forest for k' to reflect the new bindings.

The key tree adjustments are done as follows. Let v be the vertex whose function is to be changed by a *change function* graph modification step. If v is an ADD or LOOKUP vertex for key k, we remove the vertex from any key tree for k before the vertex function is changed. Then, if v is changed to an ADD or LOOKUP vertex for key k', the key forest for k' is modified to reflect the new status of v.

Removing an ADD operation for key k at vertex v corresponds to combining two base tree partitions. If v has an s-tree node for k, we must join the key tree rooted at v with the key tree above. This is done with the *join_s-tree* procedure. Adding an ADD operation for key k corresponds to splitting two base tree partitions. If v is on the path of a key tree for k, we split the key tree at v with the *split_s-tree* procedure. We discuss these procedures, shown in Algorithm 7.3.8, in Section 7.3.3.4.

A vertex v is nontrivial for key k if its vertex function is a LOOKUP operator for key k. Removing a LOOKUP vertex function for key k will result in v changing its nontrivial status for key k to trivial. The *unbind_vertex* procedure of Algorithm 7.3.6 is used to remove the base tree vertex from this previous key tree binding.

```
unbind_vertex(v : ↑ vertex)
        old_key←vertex_key(v);
        v_s←lookup(vertex_s-tree(v), old_k);
        vertex_key(v)←nil;
        if ¬ any_children(v_s) then
                split_s-tree(v, old_k);
        path_compress(v_s);

bind_on_path_vertex(v : ↑ vertex, m : integer, k : key)
        if lookup(vertex_s-tree(v), k) = nil then
                v_s←new_s-tree_map(v, k);
                if ¬ is_root(v) then
                        <p, i> ←find_next_s-tree_map(v, k);
                        p_s←lookup(vertex_s-tree(p), k);
                        c_s←s-tree_child(p_s, i);
                        graft_s-tree(p_s, v_s, i);
                        graft_s-tree(v_s, c_s, m)
```

Algorithm 7.3.6

If we change the vertex function of vertex v to a LOOKUP operation for key k', v will change from trivial to nontrivial for k'. The *bind_vertex* procedure of Algorithm 7.3.7, extended from Algorithm 5.6.1, binds a nontrivial vertex into the given key tree. The total time required to bind a vertex is $O(smod + (stest + pmod + ptest) * path + ptest * outdegree)$ where *outdegree* is the number of children at the least common ancestor base tree vertex.

```
bind_vertex(v : ↑ vertex, k : key)
        if k≠ vertex_key(v) then
                if vertex_key(v)≠ nil then
                        unbind_vertex(v);
                vertex_key(v)←k;
                if k∈vertex_on_path(v) then
                        m←find_mark(v, k);
                        bind_on_path_vertex(v, m, k)
                else
                        <lca, i> ←mark_to_next_mark(v, k);
                        if k∈vertex_on_path(lca) then
                                m←find_other_mark(v, i, k);
                                bind_on_path_vertex(lca, m, k);
                                lca_s←lookup(vertex_s-tree(lca), k)
                        else    lca_s←new_s-tree_map(lca, k);
                        v_s←new_s-tree_map(v, k);
                        graft_s-tree(lca_s, v_s, i)
```

Algorithm 7.3.7

When a vertex function is changed from a LOOKUP operation for key k to a LOOKUP operation for key k', the binding for k must be removed, and a new binding installed for key k'. Since k is stored in the *vertex_key* field of the base tree vertex, *bind_vertex* can automatically unbind the previous binding. The automatic rebinding feature of *bind_vertex* is be used in Section 7.6 where we discuss dynamic collections, which allow the key to change with out a vertex function change.

7.3.3.4 Removing or Adding an Edge

As each graph modification step alters the edges of the dependency graph, we must alter the key trees so that they are consistent with any changes in the base tree and hence the collection tree. This is done by performing the appropriate split or join operation on each of the key trees that is altered by the base tree modification. The extended *split_s-tree* and *join_s-tree* procedures are shown in Algorithm 7.3.8. The time required to split one key tree is O(*smod* + (*stest* + *pmod*)*path + *ptest***outdegree*). Joining a single key tree is done in the same time order as binding a vertex.

```
split_s-tree(v : ↑ vertex, k : key)
        if ¬is_root(v, k) then
                <p,j> ←unmark_to_next_s-tree_map(v, k);
                p_s←lookup(vertex_s-tree(p), k);
                c_s←s-tree_child(p_s,j);
                cut_s-tree(c_s);
                path_compress(p_s, k);
                v_s←lookup(vertex_s-tree(v), k)
                if v_s = nil then
                        v_s←new_s-tree_map(v, k);
                        m←find_mark(v, k);
                        graft_s-tree(v_s, c_s, m)
                else path_compress(v_s, k);

join_s-tree(v : ↑ vertex, k : key)
        if ¬is_root(v, k) then
                bind_vertex(v, k);
                v_s←lookup(vertex_s-tree(v), k);
                path_compress(v_s, k)
```

Algorithm 7.3.8

If edges are to be removed from the interior of a collection tree, many key trees may be split. In response to a *remove edge* modification step for such an edge, we locate the base tree edge that corresponds to the edge being removed. Let v be the base tree vertex at the subordinate end of this edge. Since the *vertex_on_path* set contains the keys of all key trees that pass through the edge in question, the appropriate set of key trees can be split by calling *split_s-tree* on the vertex for each of these keys. (At most, one key tree could be rooted here, if the vertex is defined by the ADD operator. This tree is not split.) The base tree can then be split using the original *split_s-tree* procedure of Algorithm 5.6.2. The total cost of removing an interior collection tree edge is $O(((stest+pmod)*path+ptest*outdegree)*psize+ptrav)$.

Figure 7.3.4 shows an example of the splitting process. Before we can remove the dotted edge in the collection tree, we must split the base tree at the vertex subordinate to this edge. This, in turn, requires us to split the key trees for a and c

that are on the path at this vertex. Note that path compression occurs both in the base tree and in the key tree for a.

If edges are added to connect a partial collection subtree rooted at v to a leaf of a collection tree, it corresponds to the connection of possibly many key trees. With the exception of the single key tree that could be rooted at v, these key trees are exactly those structure trees in *vertex_s-tree*(v). After joining the base trees using *join_s-tree* procedure of Section 5.5.3, we call the extended *join_s-tree* procedure on each of the s-trees in the *vertex_s-tree* set. Adding an edge that joins a partial collection to a leaf of another collection has a total cost of O((*smod*+(*stest*+*pmod*+*ptest*)**path*+*ptest***outdegree*)**ssize*) where *ssize* is the cardinality of the *vertex_s-tree* set at the root of the partial collection.

The joining process is simply the reverse of splitting. If we reverse the meaning of the dashed and dotted edges in the example of Figure 7.3.4, we can see how joining is done. After one adds the dotted edge to connect two collection trees, the base tree is joined causing the dashed edge to be replaced by the dotted edges. Since the key trees for a and c have s-tree nodes that no longer extend to the root of their key partitions, they must be joined. This causes the dashed edges in these trees to be replaced by the dotted ones. Note that the roots of these key trees will be removed by the *path_compress* operation that *join_s-tree* performs.

As was the case with copy trees, dependency graph modifications that insert short segments in the middle of existing collection trees do not require the division of the associated structure trees. The base tree can be extended like the copy tree was in Section 6.5. The key trees can be extended by simply duplicating the *vertex_on_path* set along the inserted segment and then correcting for ADD operations by splitting the key tree (if any) for the added key at the ADD vertices.

Figure 6.5.1 illustrates where this segment insertion occurs in attribute grammar systems. Since attribute grammar based program editors typically perform this operation at every insertion of a program statement, is important that it can be done quickly.

7.4 Propagation with Key Trees

We have constructed the key trees so that they explicitly contain the implicit identity dependencies of the collection. Instead of propagating over the inefficient original

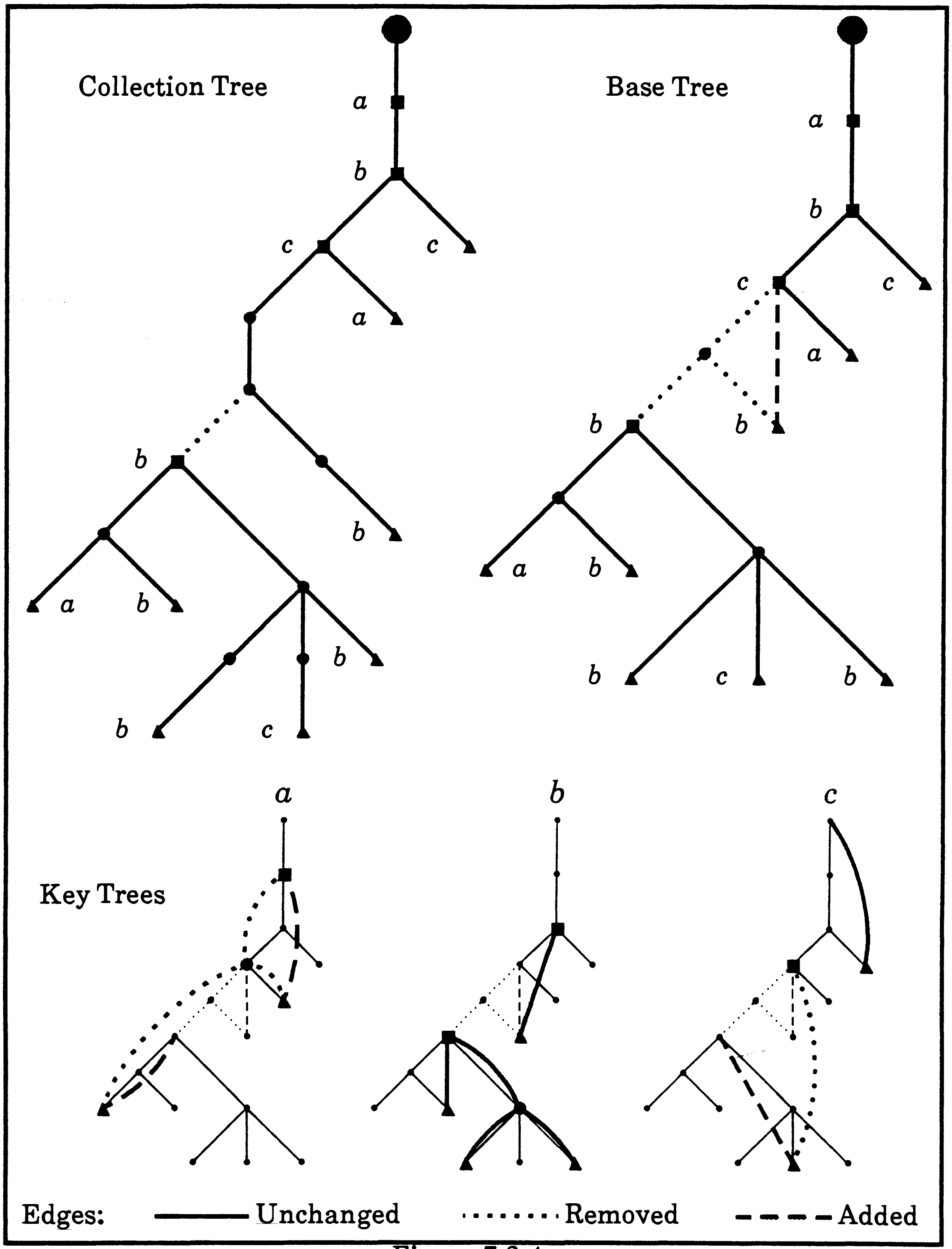

Collection Tree
Base Tree
a
b
c
Key Trees
Edges:
Unchanged
Removed
Added

Figure 7.3.4

dependency graph, we perform propagation over a dependency graph containing these explicit dependencies.

7.4.1 The Modified Dependency Graph

Let $D=(V,E)$ be the original dependency graph containing collection trees. We perform propagation over a graph D' that has each collection tree replaced by its key trees. Note that we do not explicitly construct D'. Instead, we infer D' from D and the key tree data structures. We must do a little more to construct D', however, than simply remove the collection tree and add the key trees. We need to specify how the key trees connect to the vertices in D, in order to represent the implicit identity dependencies of D.

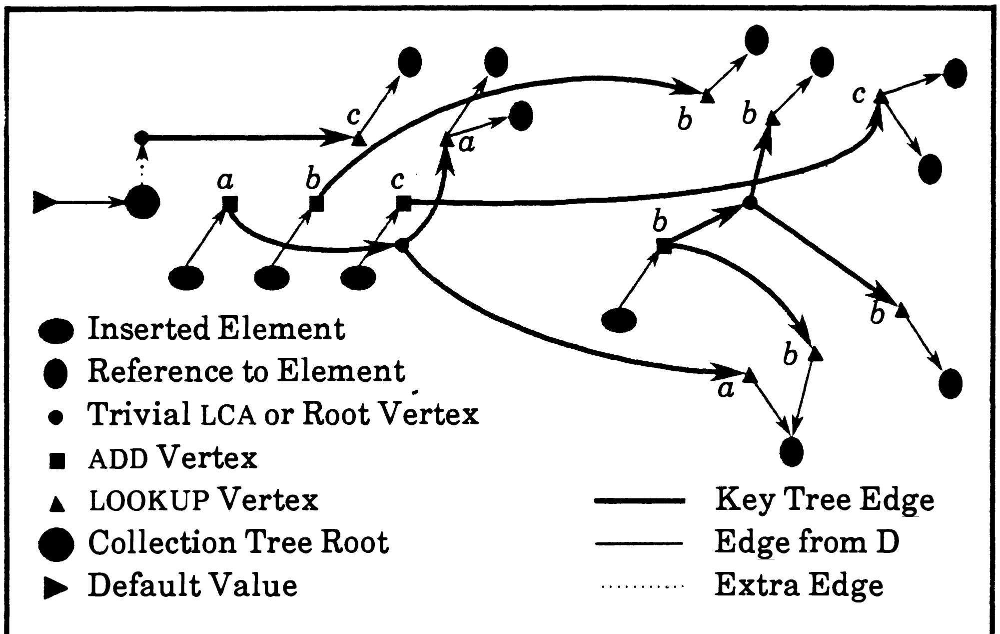
Each key tree represents a number of implicit identity dependences. Each such dependency communicates a value x from the vertex associated with the root of the key tree, to the vertex associated with a leaf of the key tree. The connections from D to a key tree, therefore, are made from the vertex that contains the value x, to the key tree root. Likewise, the leaves of a key tree will be connected to the vertices that use x as an argument.

For key trees whose root is at the collection tree root, the value to be communicated is the default value argument of the EMPTY operator. We retain the root vertex r of the collection tree in D', which is the only vertex of the collection tree retained, and connect edges from this vertex to all key tree roots at r. The vertex function of r is changed to define val_r as the default value of the collection.

The rest of the key trees, which are rooted at ADD vertices, communicate the element value specified for the ADD operators. Each ADD vertex in a static collection has two predecessors, the aggregate value and the element value. (The key is a constant specified in the vertex function. We assume that the element value is obtained from some other vertex. If it is not, an additional vertex can be generated to hold the element value). Since the ADD vertex and the aggregate value argument edge were removed with the collection tree, we are left with a disconnected edge from the inserted element value. We connect this edge to the root of the key tree.

The leaves of a key tree are associated with LOOKUP vertices. As these vertices and their predecessors are not in D', the edges that previously communicated the looked up value to vertices outside of the collection tree, remain disconnected. We connect these edges to the corresponding key tree leaf.

Figure 7.4.1 illustrates the combination of the key trees (Figure 7.2.3) with the aggregate example of Figure 7.1.2 to form the modified dependency graph D'.



Dependency Graph D' for Figure 7.1.2

Figure 7.4.1

In order to complete the definition of D', we must specify the vertex functions for the key tree vertices. As in the case of copy rules, we assign the identity function to each of these structure tree vertices. This causes the value to be broadcast throughout the key tree. The value can then be referenced at the key tree leaves by the vertices that formerly referenced the corresponding LOOKUP vertex.

Since the key tree nodes are vertices in the dependency graph, they have vertex values. We therefore must add a value field to each of the s-tree nodes that comprise the key trees. This is in contrast to the technique used to store the values for copy rule chains in Section 6.6. Since copy rule chains communicate a single value, we were able to use the value fields of the underlying vertices in D to store the communicated value. Collections, on the other hand, communicate many values, requiring us to place the values in the key trees.

Theorem 7.4.1:

Given a dependency graph D, let D' be defined as above, and let C be the class of implicit identity functions induced by collections. Graph propagation over D' can be done in time that is within a constant factor of the time required for graph propagation over $MIN_C D$.

Proof:

Similar to the proof of Theorem 6.6.1.

7.4.2 Performing Propagation

As in all graph propagation algorithms, we start with MODIFIED as the initial set of possibly inconsistent set of vertices. We construct this set by determining which of the vertices in D' could potentially become inconsistent due to each graph modification step. While these include vertices in D, they also include key tree vertices whose incoming edges have been modified. A graph propagator is then invoked on D', beginning at the MODIFIED vertices.

As we did for copy structure trees in Section 6.6, it is possible to accelerate propagation through the key trees. If a key tree vertex v changes value during propagation, all consistent key tree descendants must change value as well. Furthermore, since the vertex values for these vertices are defined by the identity function, their new values will be equal to val_v. Thus, the key tree can be traversed, communicating all of these values at once. This allows us to eliminate the vertex value comparison at each key tree vertex. During this traversal, successors of LOOKUP vertices are added to the set of possibly inconsistent vertices.

We must be careful, however, to stay within the AFFECTED region. Due to a join that may have taken place in the middle of a key tree, a vertex of a key tree may be inconsistent with its arguments. All of its descendants could already have the correct values. Updating *all* key tree descendants of v, therefore, could perform work in excess of O(INFLUENCED). To avoid this, we mark key tree vertices whose incoming edges have changed. Should the key tree traversal below v reach one of these marked vertices, u, we must compare val_u with val_v. If the values are the same, we do not traverse the subtree below. Otherwise, we continue the traversal. In either case, we unmark the marked vertex.

7.5 Key Tree Set Representation

We have left unspecified the methods of storing the two sets necessary to maintain the key trees. The *vertex_s-tree* set is used to store the key tree nodes that correspond to a given base tree vertex. The *vertex_on_path* set specifies which key trees pass through or over a given base tree vertex. In this section, we discuss several data structures that might be used to represent these sets.

7.5.1 The Vertex_S-tree Set

Four set operations are needed for the *vertex_s-tree* set. We must be able to locate elements by key, insert elements, delete elements, and traverse all of the elements of the set in some unspecified order. These requirements are met by most abstract data types that implement a *dictionary* [Aho, Hopcroft, and Ullman 74]. We will discuss the use of two of these data structures, balanced trees and hash tables.

Let m be the size of the *vertex_s-tree* set. We can store the set in a balanced tree of height $O(\log m)$. If the s-tree nodes are ordered by their keys, location, insertion, and deletion can be done in $O(\log m)$ time. Since the size of this set is bounded by the number of keys k, this implies $O(\log k)$ worst case bound on these operations. A traversal can be done in $O(m)$ time.

We can also implement the *vertex_s-tree* set as a hash table, hashing on the aggregate key values. This results in $O(1)$ expected time for location, insertion, and deletion operations. If each element in the hash table is placed on a doubly linked list, traversals can be performed in $O(m)$ time.

Both balanced trees and hash tables can be implemented in $O(m)$ space. It should be noted that there are no common elements in *vertex_s-tree*(u) and *vertex_s-tree*(v) for different base tree vertices v and u. While both of these sets may contain an element corresponding to a given key, the s-tree nodes will correspond to different base tree vertices, and will thus be different. Hence, there is no opportunity to reduce the amount of storage required by sharing portions of the data structure. Fortunately, the number of s-tree vertices used in the key trees is proportional to the size of the collection tree (Theorem 7.3.1). Therefore, the space needed is within a small constant factor of the space required by the dependency graph.

7.5.2 The Vertex_On_Path Set

Storage of the *vertex_on_path* set of size m can be done using the same methods resulting in $O(\log k)$ worst case membership, insertion, and deletion times with balanced trees, and $O(1)$ expected case membership, insertion, and deletion times using hashing. Assuming the hashed elements are placed on a doubly linked list, the traversal time is $O(m)$ in both cases.

The space required by either of these methods can not be bounded by $O(|C|)$ where C is the collection tree, since all k keys could be defined at many interior collection tree nodes. The worst case is when all definitions are close to the root, and at least one use for every key is near the leaves of C. This is not unlikely. All k keys will be defined at each of many copy vertices on a tree path above all uses and below all definitions. If we perform LOOKUP operations on the aggregate value copied at each of these vertices, the copy vertices will be a part of the base tree as well. This worst case base tree is shown in Figure 7.5.1.

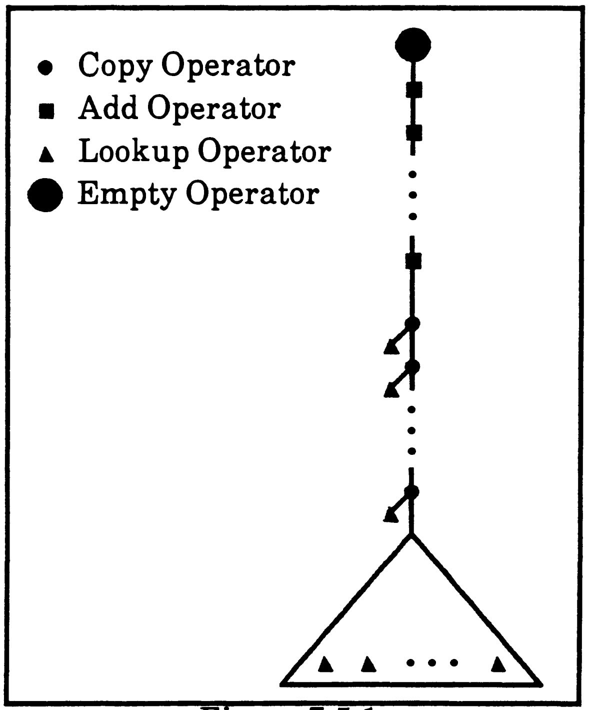


Figure 7.5.1

Since we can make the number of copy vertices, c, on this path as large as we like, c can become $O(|C|)$. Since all k keys are defined at each of these vertices, the resulting on-path set storage required by either hash tables or balanced trees is $O(|C|*k)$. As this amount of space could easily become unacceptable for large collection trees with many keys, we must find some method for reducing the required amount of storage. We will return to this problem in Section 7.5.4.

7.5.3 Execution Bounds

Using balanced trees for both data structures, we get an asymptotic running time of $O(\log k*(path+outdegree))$ for binding vertices. Splitting collection trees can be done in $O((\log k*(path+outdegree))*psize)$, and joining collection trees, in $O((\log k*(path+outdegree))*ssize)$.

These bounds, however, can be improved. Recall that we performed the splitting and joining on one key tree at a time. If we alter our split and join procedures to perform the necessary operations at each base tree vertex for all key trees at once, we can reduce the computation time.

The bulk of the time required to split key trees involves searching for the next s-tree map and clearing the on-path marks. Searching for, and setting, on-path marks comprises most of the time required for joining. If the set of key trees being split or joined, the set of s-tree maps, and the on-path sets are kept as balanced trees, we can use the merge techniques of [Brown and Tarjan 79] to locate the elements common to both trees. Thus, we can perform all the searching operations at a give base tree vertex in $O(m*\log n/m)$ where n and m are the sizes of the two sets, $n>m$. This compares to the total of $O(m*\log n)$ if each key tree is done separately, where m is the number of key trees and n is the size of the s-tree map set or on-path set that is being compared. If we extend these balanced trees to be persistent [Driscoll, Sarnak, Sleator, and Tarjan 86], we can use the above fast merge to modify the on-path marks as well.

With hash tables, the expected time bounds are $O(path+outdegree)$, $O((path+outdegree)*\text{psize})$, and $O((path+outdegree)*\text{ssize})$ for binding, splitting, and joining respectively.

There are other issues, however, that we must consider in our choice for the *vertex_on_path* set data structure. We discuss these factors in the remainder of Section 7.5.

7.5.4 Data Sharing in the On-Path Sets.

As mentioned in Section 7.3.2, it is possible, through the use of persistent data structures, to allow the on-path sets to share parts of their data representation when constructing the initial key trees. However, this is not sufficient to retain sharing after a large number of small changes to these sets. Since we are primarily interested in accelerating incremental evaluation, the overhead associated with these methods would outweigh any advantages.

Another approach to sharing on-path data is to have the same key sets represented by the same data. By storing a pointer to an on-path set, rather than the set itself, we could have two vertices with equal on-path sets sharing to the same set data structure. A reference counting scheme would allow sets to be removed when all pointers to them have been removed.

Since maintaining the invariant "all equal sets are represented by the same data structure" is somewhat costly, we settle for a weaker invariant.

Invariant 7.5.1:

> If two base tree vertices, u and v, are adjacent in the tree, and if the set *vertex_on_path*(u) is equal to the set *vertex_on_path*(v), then the same storage is used to represent the sets.

Each on-path set, therefore, will be associated with a contiguous set of base tree vertices.

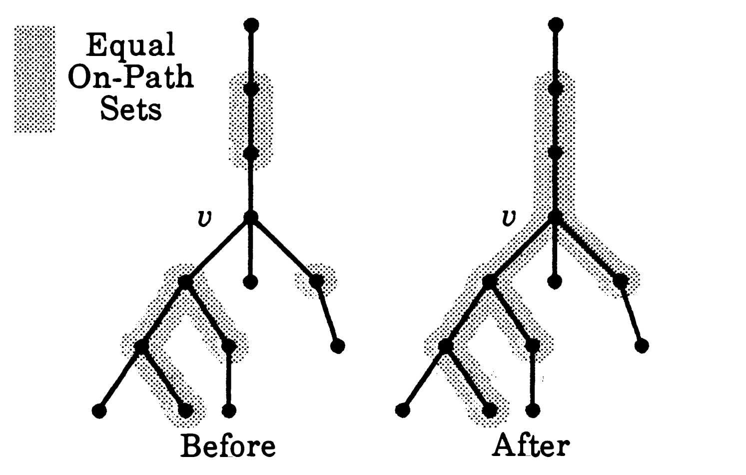
Definition 7.5.1:

> Let S be an on-path set. We call a maximal connected set of base tree vertices whose on-path sets are represented by S, the *representation set* for S.

Since each base tree vertex v has a single on-path set, v will be in one, and only one, representation set. While we could represent the *vertex_on_path* field of v as a pointer to the on-path set, efficient maintenance of Invariant 7.5.1 requires a slightly more complicated method.

7.5.5 Maintaining the Invariant

Invariant 7.5.1 could be invalidated any time that we add or remove a key from the on-path set for any vertex v. Let A be the set of vertices adjacent to v. The new on-path set for v could be the same as the on-path set of a vertex $u \in A$, requiring us to add v to the representation set for u. In addition, for each $w \in A$, $w \neq u$, the representation set containing w should be combined with the representation set for u if *vertex_on_path*(w) = *vertex_on_path*(u). Figure 7.5.2 illustrates this situation. We must also perform a union of the representation sets for u and w if an edge is added between u and w and the on-path sets for these vertices are equal.



Changing the On-Path Set for v

Figure 7.5.2

Performing these set unions with a simple pointer representation would require that we traverse the smaller representation set to update the pointers when performing a set union. This would require $O(m)$ work for each union, where m is the size of the smaller set. We can substantially reduce the work required, by using the fast disjoint-set union algorithm [Aho, Hopcroft, and Ullman 74] [Tarjan 83]. By maintaining the representation sets this way, we can perform any sequence of n *union* and *find* operations in $O(n^*G(n))$ time, where G is roughly the inverse Ackermann function. Find operations allow us to locate the on-path set of a vertex, and union operations allow us to combine representation sets. Since $G(n)$ is essentially constant for all practical purposes, we can treat the cost of each operation as an amortized O(1).

We also need to know when the on-path sets for two adjacent vertices are equal. While a comparison of the set sizes, and perhaps the comparison of a hashed value maintained of all of the keys in the sets, would make most unequal sets known in constant time, a true comparison of the keys, necessary to verify equal sets, would take time proportional to the size of the sets. We can reduce this worst case cost with the following scheme.

At each vertex v we store an integer *difference count* of the number of keys that appear in exactly one of the on-path sets of v and its parent. This value is undefined at the root of the base tree. When the difference count at v is zero, the on-path sets of v and the parent of v must be equal. By maintaining the difference counts at the vertices of the base tree, the equality test of the on-path sets can be avoided.

Before we add or remove a key from the on-path set for v, we check the on-path sets of the adjacent vertices, and adjust the difference counts of v and the children of v. For most dictionary structures, this increases the on-path set modification time to $O(outdegree*ptest+pmod)$, where *outdegree* is the number of children of v, *ptest* is the time required to test if an on-path set contains the given key, and *pmod* is the former on-path set modification time.

Let Z be the set of all children of v whose difference counts go to zero, plus the representation set of the parent of v if the difference count at v goes to zero. If $Z=\varnothing$, we change the on-path set for v (see Section 7.5.6). Otherwise, let R_Z be the set of representation sets that contain elements of Z, located with the find operation. We remove v from its representation set and combine it with all representation sets in R_Z, using the union operation. This maintains the invariant.

Since there is no difference count between vertices connected by a newly added edge, a true comparison between the on-path sets must be performed at these vertices. A difference count for the subordinate vertex is computed during the comparison.

7.5.6 Changing the On-Path Sets

As we split, join, and bind vertices to key trees, we modify edges of the key trees. When this is done, we update the on-path sets for vertices that are on the base tree path p that is represented by the modified key tree edge. These modifications involve adding and removing keys from the on-path sets. Since the on-path sets are potentially shared by a number of vertices, we cannot blindly change the on-path data structure. If we did, we might incorrectly alter the on-path set of another vertex.

If we are storing the on-path sets using balanced trees or some other linked structure, a persistent data structure [Driscoll, Sarnak, Sleator, and Tarjan 86] could be used at the expense of an additional amortized O(1) factor in the time and space required per update. For other data structures such as hash tables, we must copy the entire on-path data structure to gain persistence. This copying operation requires, in general, $O(m)$ time and space, where m is the number of elements in the on-path set.

At the cost of increasing the on-path set operation time by a constant amount, we can decrease the amount storage and time required for copying by a constant factor. This is done by providing a constant sized overflow area for keys in on-path sets as shown in Figure 7.5.3. With this modification, an on path set is now a pointer to the old on-path set plus a pointer to the overflow area. When a persistent update is required, we make a new pointer to the old on-path data structure and copy the overflow area, adding our new change. Any element testing must first search the overflow area. When the overflow area is full, a subsequent modification to a key not in the overflow area will force the former on-path data structure to be copied. When this is done, all modifications in the overflow area are made to the data structure

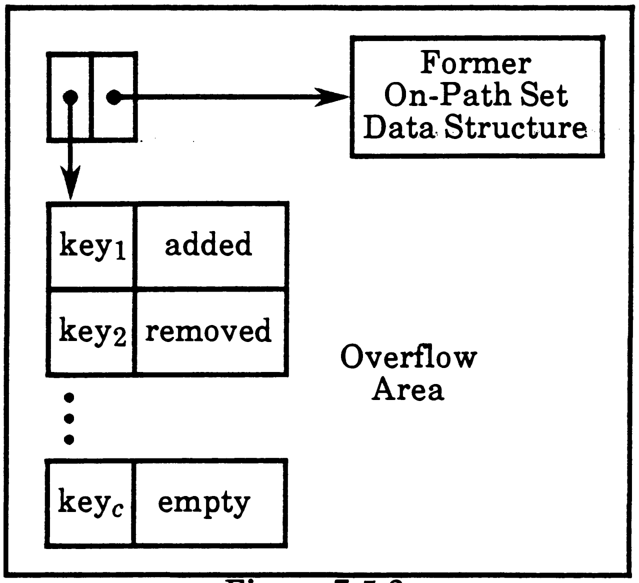


Figure 7.5.3

Some updates, however, permit us to destructively update an on-path set S. If the entire representation set for S is on path p, we can update all of their on-path sets at once by destructively changing S. We can determine whether or not this is the case by traversing p and counting consecutive vertices that are in the same representation set. If this number is equal to the reference count at the corresponding on-path set, a destructive update is permitted. In addition, this allows us to avoid recomputing the difference counts at all vertices interior to p.

Even if the entire representation set is not on p, we can perform the on-path set change on a single vertex v, and then add to v's representation set all vertices on p that are in the former representation set of v. Thus, each representation set intersecting p requires at most one persistent update.

7.5.7 Decreasing the Number of Different On-Path Sets

We note that it is possible to limit the number of different on-path sets by requiring that all key trees extend to the base tree root. This is achieved by simply binding the ADD vertices for key k into the key tree for k above them. The result is a single key tree per key. Key tree edges to ADD vertices are ignored with respect to propagation and vertex function evaluation. Extending the key trees in this way places a bound on the number of on-path sets on a given path from a leaf to the root in the base tree.

Theorem 7.5.1:

Let p be a path from a leaf of the base tree to the base tree root, and let K be the set of keys used in the collection. The number of unequal on-path sets on path p is $O(|K|)$.

Proof:

Since all key trees extend to the base tree root, $k \in vertex\_on\_path(v)$ implies that $k \in vertex\_on\_path(\text{parent}(v))$. Therefore, the on-path set for the parent of v contains the on-path set for v. Since the on-path set at the root of p contains at most $|K|$ elements, there can be at most $|K|$ different on-path sets along p.

This technique, in combination with the sharing method and overflow area discussed earlier, allows us to reduce the on-path storage for the example of Figure 7.5.1 to $O(|K|^2 + b)$ space, where b is the size of the base tree. The sharing reduces the number of on-path sets, each of which is $O(|K|)$ in size, to $O(|K|)$. The overflow area allows us to store the on-path sets on short branches off of the main path in $O(1)$ space.

While extending all trees to the base tree root seems somewhat extreme, we note that in a definition-use system that requires multiple definitions to be detected, such as variable environments in Pascal, the resulting collection effectively does this. Since, as shown in Figure 7.5.4, each ADD operator for key k must be preceded by a LOOKUP operator for k to test for the default value of the collection, the on-path sets on the path to the root are the same as if we connected all key trees for k.

7.5.8 Bit Vector Representation

Another factor that affects our choice of the on-path data structure involves the type of information represented by the set. Since we are representing a boolean variable that specifies the on-path status for each key, a single bit for each key would suffice. While this increases the storage cost of each on-path set to $O(k)$, where k is the number of keys, the constant is very low compared to balanced trees or hash tables. If the number of keys is small, or we expect most on-path sets to be somewhat dense (as in Section 7.5.7), bit vectors will require less time and space to maintain.

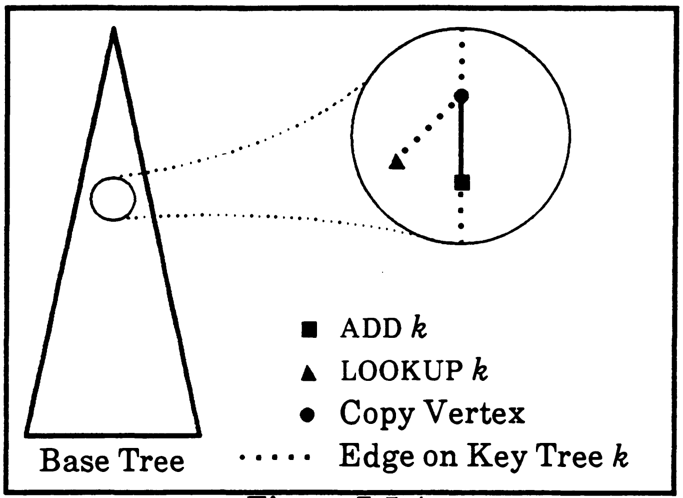


Figure 7.5.4

In order to use a bit vector, we must first translate the keys used in the collection (and any other collection to which it might later be connected) into small integers. This can be done as follows. We initialize a key counter, *kc*, to 0. The first time a key is encountered, we assign it the value of *kc*, and then increment *kc*. We place the value in a hash table, hashed on the key. Subsequent uses of a key will obtain the integer value from the hash table. Reference counting of all references to each integer key, along with a free key list of no longer referenced integers, allows us to reclaim the numbers that have been assigned to keys no longer in use. The bit vectors, therefore, can be kept somewhat dense. The translation of keys into integers is a good idea, regardless of the on-path set representation, since it allows faster comparison times over more complicated keys. Key comparison can become a major expense in any dictionary structure that performs a large number of comparisons, such as balanced trees.

Thus, our keys are now small integers. A modification to an on-path set for key i, therefore, is simply a matter of changing the i^{th} bit in the bit vector. The sharing techniques of the previous sections, including the overflow area, can be used to reduce the storage costs. We keep a length count on each bit vector which allows us to eliminate zero bits at the end of a vector—changes to these bits are stored in the overflow area. When vector is copied, it is extended to hold the overflow keys. This also allows new keys to be introduced into the collection in a consistent fashion.

7.6 Dynamic Collections

In Section 7.3 we discussed how the base and key tree data structures can be maintained for static collections. In this section, we extend these algorithms to dynamic collections.

The primary difference between static and dynamic collections is that the keys used in dynamic collections can be computed by the dependency graph. Since we do not know the values assigned to these keys until propagation time, the key trees cannot be built or updated before propagation begins.

In the introduction to this chapter, we explained why the straightforward implementation method for aggregates was inefficient—propagation assumed all of the uses in the aggregate to be dependent upon all of the definitions. In the static case, we could eliminate most of these dependencies from the dependency graph after a static analysis of the collection. In the dynamic case, we cannot decide (given arbitrary vertex functions) whether or not a given use is affected by a given definition without evaluating the vertex functions of the AFFECTED set. Thus, all uses are, in general, dependent upon all definitions.

If a substantial number of the keys used in the collection change at every incremental update, our efforts to determine the implicit identity dependencies will be in vain. The set of such dependencies would be drastically altered at each incremental change. Maintaining explicit dependencies between definitions and uses would require substantially more time than the simple technique of passing a set of bindings throughout the collection tree at every update. Therefore, one should make the implicit identity dependencies explicit, only if the set of keys used is somewhat stable. We will assume that at most a small number of keys, if any, change after each dependency graph modification.

We use a similar approach for dynamic collections to the method used for static collections. The major difference is that we cannot update the key trees at the same time that the base tree and the underlying dependency graph is changed. Instead, we must wait until propagation time to update the key trees.

Before the initial evaluation, all LOOKUP vertices are not bound to key trees—we bind them into the appropriate key tree when evaluation has determined their key. During the initial evaluation of the dependency graph, the vertices are visited in a topological order of the original dependency graph D. As we reach each LOOKUP vertex v, the *vertex_key* field for v will be *nil*. Since we will have computed the correct key value of the lookup, we can bind this vertex into the proper key tree, and obtain the value being looked up. The key value is then stored in the *vertex_key* field for v.

Upon subsequent evaluations of v, we must verify that the value of the key has not changed. If the key stored in the *vertex_key* field is not equal to the current key for the LOOKUP operation, we must remove v from the key tree specified by *vertex_key* and place it in the new key tree. The *bind_vertex* procedure of Algorithm 7.3.7 is written to perform this automatic rebinding operation. Therefore, we call *bind_vertex* as propagation reaches each LOOKUP vertex. If the key remains the same, the operation requires constant time.

If an ADD vertex changes key, we must join the key tree (if any) for the old key, and split the key tree (if any) for the new key (see Section 7.3.3).

Since we perform the incremental evaluation by propagation over the key trees in the modified dependency graph D' (Section 7.4.1), we could reach the affected ADD and LOOKUP operations through propagation in the wrong key tree. Since the key trees are independent of one another, the correct value may not yet have propagated to the new key tree. Thus, the value obtained for the LOOKUP vertex could be wrong. Likewise, if propagation has reached all of the vertices in the new key tree, the value defined by the ADD vertex could be different. While in both cases, propagation insures that the final values will be correct, a large amount (not bounded by AFFECTED) of unnecessary work could have been done while assigning these vertices, and their successors, incorrect values.

This excess work is due to the fact that the true dependencies are not reflected in D'. If the keys are very stable, we may be able to ignore the cost of this excess work. Otherwise, we must force propagation to proceed in the order of the original dependency graph D. We do this by including the collection tree edges in D'. While we ignore these edges when evaluating vertex functions and expanding the possibly inconsistent vertex set, these edges will force evaluation to occur in the order of the collection tree. This guarantees that no LOOKUP vertices will be evaluated before their definitions.

Note that dynamic collections do not quite fit the model that we have developed for incremental evaluation. Since key tree edges are part of the dependency graph, and since we are modifying the key trees during propagation, we are performing graph modification operations during propagation. This blurs the sharp division between dependency graph modification and propagation that we presented in Chapter 2.

While most of the graph propagation algorithms of Chapter 3 can be extended to allow modification during propagation, maintaining an exact topological order of the dependency graph can become expensive. For example, we must adjust the order of the vertices in the priority queue of the maintained topological order algorithm (Section 3.8). In Section 8.3, we discuss how approximate topological ordering can be extended to allow these concurrent dependency graph modifications.

7.7 Nested Collections

While any combination of definitions and uses can be implemented using collections, the absence of an ability to combine two collections in order to form a combined collection can make the construction of the dependency graphs somewhat awkward. The ADD operator forces definitions to be added one-by-one, preventing multiple definitions from being added to the aggregate at a given vertex. Thus, large numbers of definitions tend to shape the collection tree into a long linear list of definitions terminated by a tree of uses. This implies that the key trees, which represent the communication between these definitions and uses, will have similar structure.

Consider the program P_0 of the ALGOL family shown in Figure 7.7.1. This program has a set of variable declarations, variable uses, and two lexically nested procedures, P_1 and P_2. Each of these procedures have a set of local declarations and variable uses.

We can implement the variable environment for program P_0 using collections by threading a collection through the declaration sections of the program and procedures. Figure 7.7.2 shows an example of how this might be done. Notice how the definitions in the interior scope of P_1 have all definitions of P_0 above them in the collection tree.

A major problem with this method is the difficulty of testing for multiple declarations within a scope. Without storing additional information (such as the scope number), it is impossible to differentiate a definition added previously in the current scope from one added in an enclosing scope. If the scope number is stored, the use sites that detect previous definitions will get bound to key trees that extend to the previous definition in an outer scope, or to the root of the collection if there is no such definition. This causes excess searching in structure tree operations.

P_0

$decl_0[1], decl_0[2], \ldots, decl_0[d_0]$

P_1

$decl_1[1], decl_1[2], \ldots, decl_1[d_1]$

$use_1[1], use_1[2], \ldots, use_1[u_1]$

P_2

$decl_2[1], decl_2[2], \ldots, decl_2[d_2]$

$use_2[1], use_2[2], \ldots, use_2[u_2]$

$use_0[1], use_0[2], \ldots, use_0[u_0]$

Program P_0

Figure 7.7.1

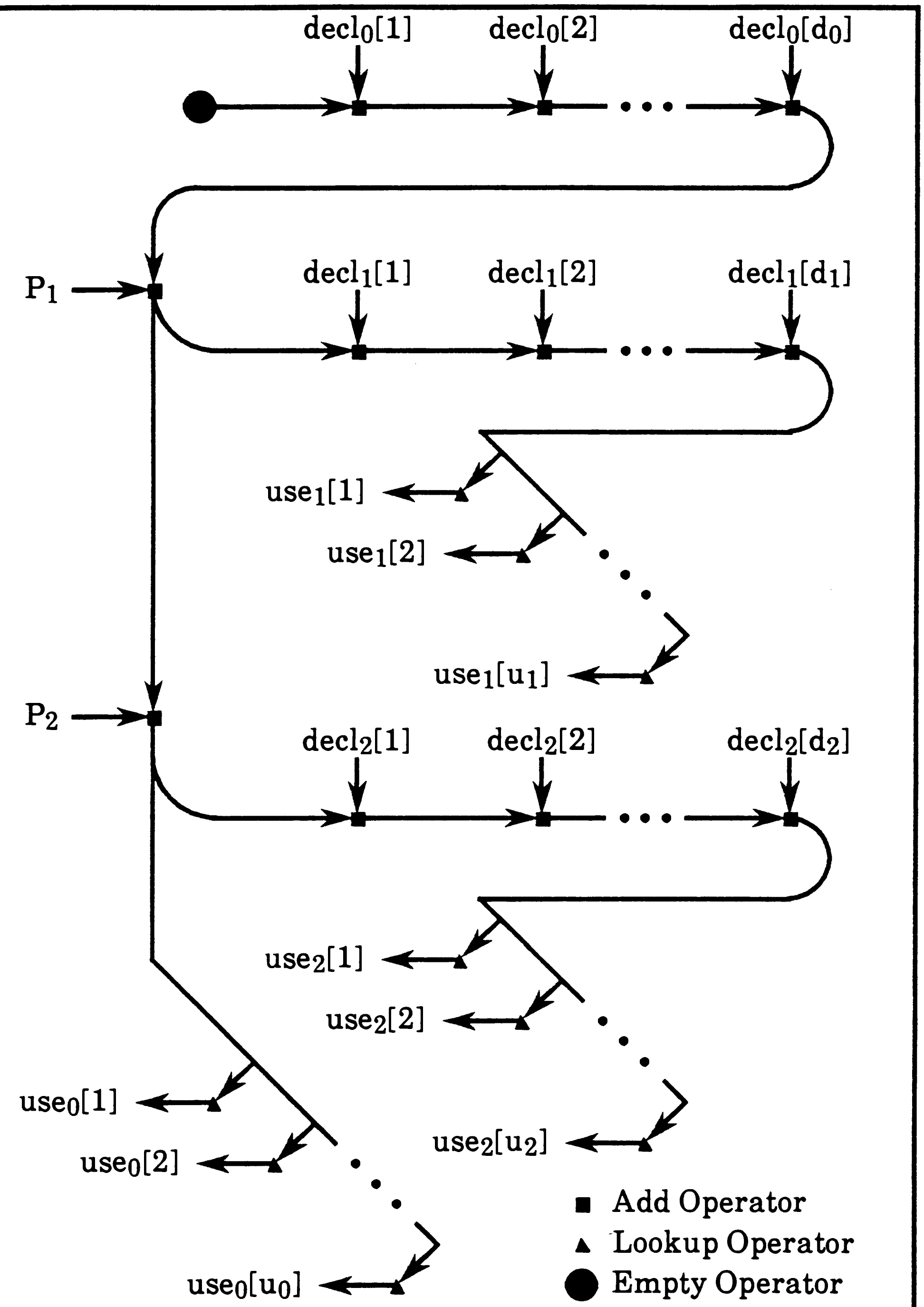
We solve these problems by using the UPDATE operator to combine multiple collections, one for each scope. We refer to the aggregates that can be formed in this way as nested collections.

Definition 7.7.1:

A *nested collection* is an aggregate that is defined using the EMPTY, ADD, LOOKUP, COPY, and UPDATE operators.

Figure 7.7.3 shows how the variable environment of P_0 might be computed using a nested collection. The local environment of each procedure is constructed as a separate collection. The combined environment is computed at the entry of each procedure block by applying the UPDATE operator to the enclosing environment and the local environment.

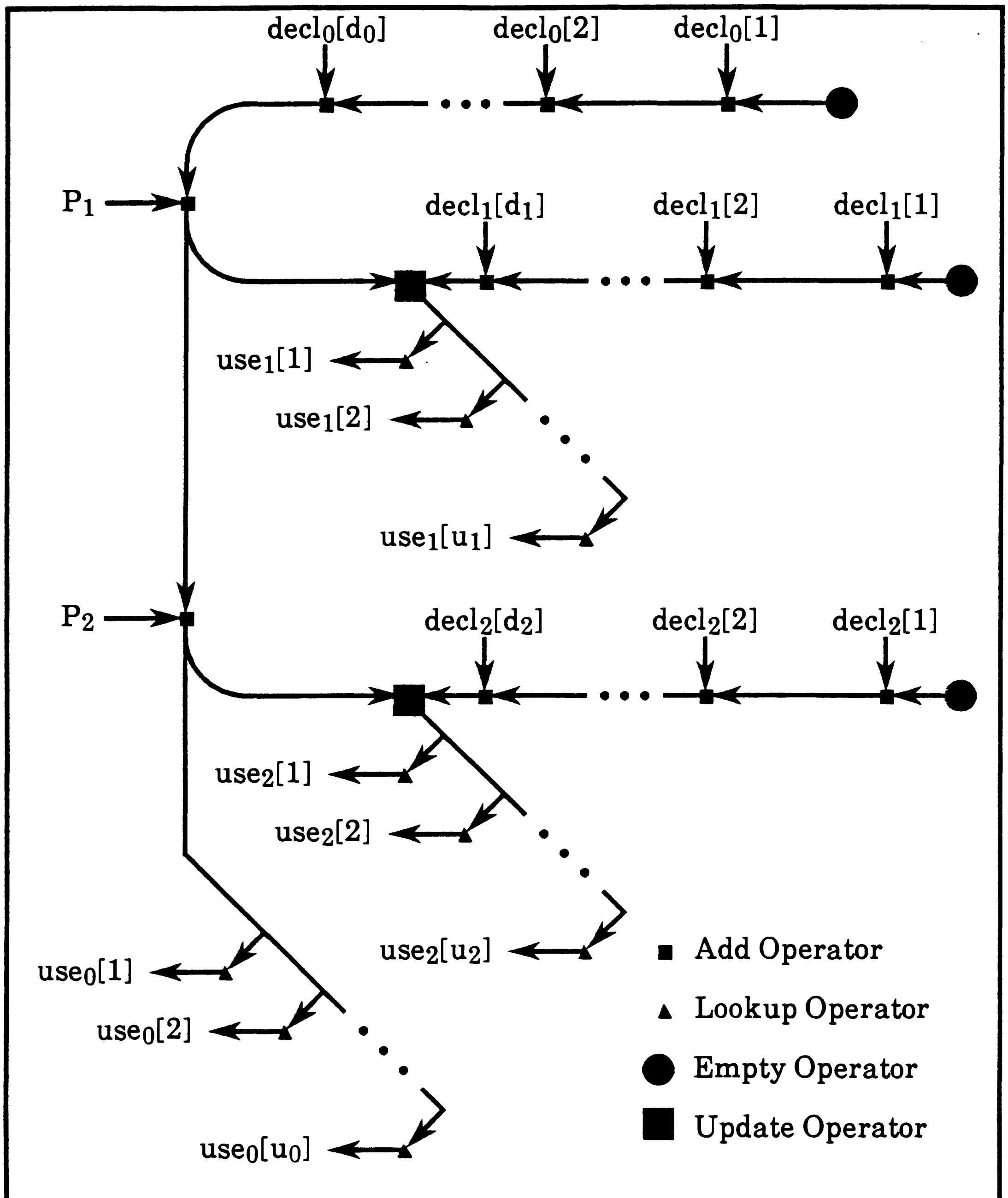
The nested collection method eliminates long chains of definitions for nested scopes. However, it also complicates the task of explicitly representing the implicit identity dependencies. In this section, we show how to communicate defined values between definitions and uses in different collection trees that are connected through the use of the UPDATE operation.



Collection Environment Implementation for Program P_0

Figure 7.7.2

We consider the UPDATE operator to be a leaf of both collection trees that it is in, as well as the root of the collection tree that it defines. While we maintain nested collections using similar data structures as those used for unnested collections, we must also maintain dependencies between definitions and uses that cross scope



Nested Collection Environment Implementation for Program P_0

Figure 7.7.3

boundaries. As we did in Section 7.3, we incrementally maintain a base tree for each collection tree, and key trees for each key used in the collection. Nested collections differ, however, at aggregate vertices defined by the UPDATE operator. Each such vertex can potentially be part of three collection trees. Figure 7.7.4 shows collection C_2 defined as UPDATE(C_0, C_1).

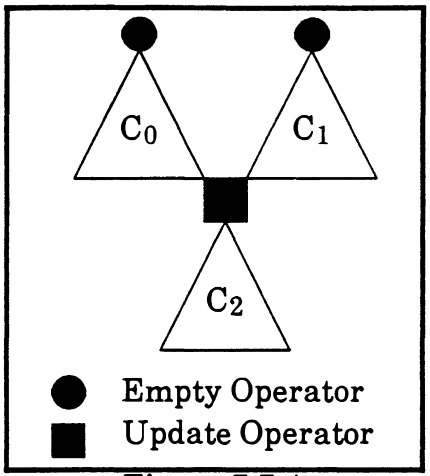


Figure 7.7.4

We must insure that correct communication occurs between the definitions and uses of different scopes. We do this by explicitly representing dependencies between references to the same key that are located in different, nested collection trees. We maintain the consistency of these explicit dependencies in the presence of dependency graph modifications.

The data structure used to connect the scopes is discussed in Section 7.7.1. In Sections 7.7.2 and 7.7.3, we extend the *bind_vertex* procedure of Algorithm 7.3.7 and the *split_s-tree* and *join_s-tree* procedures of Algorithm 7.3.8 to maintain this data structure. Propagation between scopes is discussed in Section 7.7.4.

7.7.1 UPDATE Vertex Data Structure

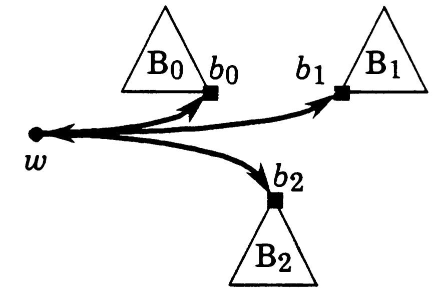
Since UPDATE vertices can be in three collections, they can also be in three base trees. Therefore, instead of one base tree vertex at UPDATE vertex u, there are three base tree vertices, one for each collection. To represent them, the *vertex_s-tree* and the *vertex_on_path* fields of the underlying collection tree (Figure 5.3.1) must become arrays of three elements. We will store the base tree vertex for the left-hand UPDATE parent in element 0, the right-hand UPDATE parent in element 1, and the combined base tree in element 2. The UPDATE vertex fields are shown in Figure 7.7.5

```
update_vertex : record of
        vertex_s-tree : array 0 .. 2 of ↑ s-tree;
        vertex_on_path : array 0 .. 2 of boolean;
        ... other vertex fields ...
```

Figure 7.7.5

While the underlying collection tree vertices with UPDATE vertex functions are different from other vertices, the base tree vertices (Figure 7.3.1) remain unchanged. In addition, the key trees remain the same within a given base tree.

Let B_0, B_1, and B_2 be the base trees for collections C_0, C_1, and C_2, and let b_0, b_1, b_2 be the left-hand, right-hand, and combined base tree vertices respectively. The mapping between the collection UPDATE vertex w and the three base tree vertices is illustrated in Figure 7.7.6.



UPDATE Vertex to Base Tree Mapping

Figure 7.7.6

If a LOOKUP vertex v in base tree B_2 references a key k that is not defined above v in B_2, the value of the definition of key k is defined to be the last value for k added to the collection in B_1, or the value defined for k in B_0 if a value for k is not added in B_1. In order to represent the communication of the value for key k from B_0 or B_1 to B_2, we maintain the following invariant.

Invariant 7.7.1:

- If v's value is obtained from B_0, then base tree vertices b_0, b_1, and b_2 have s-tree nodes for k.
- If v's value is obtained from B_1, then base tree vertices b_1 and b_2 have s-tree nodes for k, while b_0 does not.
- Otherwise, v's value is obtained from B_2. Neither b_0, b_1, nor b_2 have s-tree nodes for k.

Since collection C_1 takes precedence over collection C_0, we keep b_1 nontrivial in B_1 even when no definition for key k exists there. Should a definition for k be inserted in the right-hand collection tree at some later time, the new definition will become a part of the key tree for k in B_1. It is the existence of this key tree for k in B_1 that allows us to determine that there are uses of the definition below.

7.7.2 Binding Vertices in Nested Collections

When we bind a vertex v for key k in a given base tree B_2 (*bind_vertex*, Algorithm 7.3.7), a key tree for k is extended from the appropriate definition of k, or from the root of B_2 if there is no such definition (*bind_vertex*, Algorithm 7.3.7).

If a definition for k is found inside B_2, there is no communication between scopes to be represented. The key trees for k are identical to those in collections without the UPDATE operator.

Otherwise, k is undefined in B_2, and the key tree for k is extended from the root b_2 of base tree B_2. Let w be the collection tree vertex that corresponds to b_2. *Bind_vertex* will insert a key tree node for key k in the *vertex_s-tree* set of b_2. If w is an EMPTY vertex, there is no enclosing scope. If w is defined by the UPDATE operator, however, b_2 will be given an s-tree node for k, invalidating Invariant 7.7.1. This indicates that there may be a value defined for k in an enclosing scope. We search the enclosing scopes for such a definition, binding b_1 for k. If no definition is found on the right, we bind b_0 for k. This process restores Invariant 7.7.1.

Note that when binding one of the parent base tree vertices, a recursive invocation of the process may occur if the parent collection tree is itself rooted at an UPDATE vertex. Also, if there is no definition for k in any enclosing scope, the use inherits the default value of the EMPTY vertex at the root of the leftmost enclosing scope. The *nested_bind_vertex* procedure is shown in Algorithm 7.7.1.

The bindings of vertices can be removed using a similar traversal of enclosing scopes. In each scope, *unbind_vertex* (Algorithm 7.3.6) is called to remove the vertex from its key tree. If this causes the key tree to be removed, and the key tree is rooted at an UPDATE vertex, the root vertex must be unbound from the parent collections.

7.7.3 Splitting and Joining in Nested Collections

The splitting and joining of key trees is identical to unbinding and binding vertices with respect to the scope boundaries *above*—splitting causes the scope bindings to be removed up to the next reference to the given key k, joining causes the scope bindings to be extended up to the next reference k. Maintaining Invariant 7.7.1, however, may

```
nested_bind_vertex(v : vertex, k : key) : boolean
        bind_vertex(v, k);
        if key tree was not extended to base tree root then
                return(true)
        else if base tree root is EMPTY vertex then
                return(false)
        else
                w←UPDATE vertex at base tree root;
                found_it←nested_bind_vertex(right_base_tree(w), k);
                if ¬found_it then
                        found_it←nested_bind_vertex(left_base_tree(w), k);
                return(found_it)
```

Algorithm 7.7.1

require us to adjust the bindings in scopes *below* the split or join. This can happen as follows.

Say that a collection tree C is split, and C is between a definition for key *k* in collection tree D and use for key *k* in collection tree U. Since D no longer encloses U, we no longer have a definition connected to the use. There might, however, be another definition for *k* in an enclosing scope of U in a left parent scope of any of the UPDATE operators between C and U. We bind the base tree vertices for these collections to *k* at these UPDATE operators until a definition is found. In the example of Figure 7.7.7, C_1, and then C_2, must be bound. If no definition exists, the default value for C_2 will be communicated to the use site.

When joining collection tree C, a definition might appear above that supersedes a definition on the left in an UPDATE operation below. If we find a definition for key *k* above when we join the key tree for *k*, we must remove the bindings of C_1 and then C_2 (if C_2 is bound) to restore Invariant 7.7.1.

The above algorithms assume that we can easily locate the UPDATE vertices below a given point in the base tree. In order to efficiently find the UPDATE operators within a given scope, we specify a reserved key for updates, called the *update key*, and bind all UPDATE vertices to this key. This creates a key tree in each scope containing the update operators used in the given scope. We call this key tree the *update tree* and use it to navigate through the various nested collections. During incremental

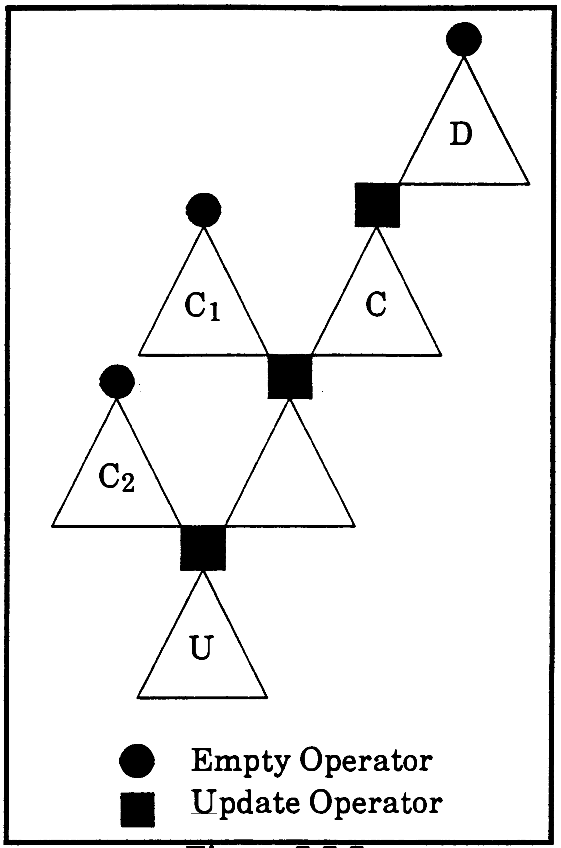


Figure 7.7.7

modification steps, the update tree is maintained in exactly the same way as any other key tree.

To locate the dependent UPDATE vertices, the on-path set at v is examined for the reserved update key. If the update key is in this set, there is at least one UPDATE operator that is dependent upon v. The *find_next_s-tree_map* procedure (Algorithm 7.4.4) can then be used to locate the previous s-tree node in the update tree. Following the key tree edge that is indicated by *find_next_s-tree_map* will locate a subtree of affected UPDATE operators. A traversal of this subtree will find each base tree vertex v that corresponds to an UPDATE vertex w.

If v is the left-hand base tree vertex, the UPDATE operation is not affected by the change above. Otherwise, v is the right-hand base tree vertex at w. We bind or unbind the left base tree vertex at w, depending upon whether we are splitting or joining. The same traversal is then recursively applied on the update tree for the base tree rooted at the combined base tree vertex of w.

7.7.4 Propagation Through Update Operators

When performing propagation through a key tree for k, the graph propagator may encounter key tree vertices that correspond to UPDATE vertices. By Invariant 7.7.1, the combined base tree vertex, the right base tree vertex, and possibly the left base tree vertex, are bound to key k. To allow propagation between the corresponding key tree vertices, the modified dependency graph D' (Section 7.4.1) must be extended.

Each key tree vertex corresponding to an UPDATE vertex is given the identity function for its vertex function. Edges are maintained in D' to connect the vertices as follows. If all three vertices are bound, an edge is added to connect the left base tree vertex with the combined base tree vertex. Otherwise, an edge is added to connect the right base tree vertex with the combined base tree vertex.

The above dependency graph construction communicates the value through the intermediate update operators to the dependent uses. Each scope adds an extra O(1) step to the time required to propagate between definitions and uses.

7.8 Relationship to Previous Work

The idea of connecting definitions with uses is not new. [Johnson and Fischer 82] constructed lists of uses for each identifier. In an attempt to formalize this approach, [Johnson and Fischer 85] extended the attribute grammar to allow *nonlocal productions*. While the aggregate value is still passed through the dependency graph, these productions allow dependencies to be constructed between definitions and uses. While this allows changes in the value bound to a given definition to be directly propagated to the use sites, additions to or deletions from the aggregate still require every vertex in the dependent portion of the aggregate to be evaluated.

Other researchers have applied conventional symbol table data structures to this problem. Hand coded symbol table modules are used to manage incrementally evaluated environments by [Staudt, Krueger, Habermann, and Ambriola 86] and [Reiss 84]. [Beshers and Campbell 85] attach procedures to each definition site. The procedures are responsible for constructing and maintaining the definition in a symbol table.

[Reps, Marceau, and Teitelbaum 86] sketch how their method for bypassing upward remote references can be extended to limit the propagation through aggregates in attribute grammars. The uses are stored in a tree structure and are ordered by their appearance in the abstract syntax tree. Changes in definitions are then limited to the portions of the tree where their uses are located.

[Hoover and Teitelbaum 86] first exploited the fact that the aggregate structure can easily be determined if it is be constructed with a set of primitive operations. A structure similar to key trees, though less efficient, was used to maintain the uses of each value defined in the aggregate. The definitions, however, were not similarly maintained. A change in a definition, therefore, caused all dependent definitions to be evaluated, regardless of their key. The key tree method presented in this chapter eliminates this problem

8. Approximate Topological Ordering

8.1 Introduction

We would like to be able to restore a dependency graph to consistency after an arbitrary graph modification operation in time close to O(|INFLUENCED| + EVAL(INFLUENCED)). As was discussed in Chapter 3, it is not known whether or not an algorithm exists that runs in this time bound. In this chapter, we describe Approximate Topological Ordering [Hoover 86], an approximate algorithm that has yielded experimental results close to this bound in cases that have arisen in practice.

This propagation technique has been implemented and successfully used in an attribute grammar based incremental editor system, the Synthesizer Generator [Reps and Teitelbaum 84]. The dependency graphs evaluated were those specified by various attribute grammars, in addition to dependency graphs with copy rule chains [Hoover 86] and aggregate communication [Hoover and Teitelbaum 86] replaced by direct communication. Averaged over an extended number of incremental evaluations, the number of unnecessary vertex evaluations due to the approximate nature of the ordering is typically several percent of the total number of vertex function evaluations.

In Section 8.2, we examine the algorithms for approximate topological ordering. Because the algorithm assumes so little, it can be extended to work under variety of conditions. In Section 8.3, we show how it can be used for concurrent dependency graph modifications. We discuss parallel evaluation in Section 8.4.

8.2 The Algorithm

Our approach is to use a heuristic that approximates the topological order of the vertices in the possibly inconsistent vertex set. It does not stay within the O(|INFLUENCED|) worst case time bound for the bookkeeping required during propagation. In fact, it does not even keep the O(EVAL(INFLUENCED)) bound on the time required to evaluate the vertex functions. We cannot argue worst case optimality for this algorithm. Instead, we claim that in practice, for some dependency graphs and dependency graph modifications, the time required is close to O(|INFLUENCED| + EVAL(INFLUENCED)).

8.2.1 The Heuristic

Let W be the set of possibly inconsistent vertices. We approximate the topological order of the vertices in W as follows.

When the initial evaluation of the dependency graph is performed, each vertex is assigned an *order number*. These order numbers are constructed to reflect a true, sparse, topological ordering of the graph. If the dependency graph is evaluated in topological order, as suggested in Chapter 2, we can obtain this numbering by simply counting the vertices as they are visited. A sparse numbering is obtained by appending a fixed number of digits to this count.

When the graph is modified, we perform the specified modifications and do nothing to the order numbers. In general, after performing the modification, the ordering is no longer correct.

We choose the element of W with the lowest order number as the vertex to be evaluated first. Since the ordering might no longer be a true topological ordering, we are not guaranteed that this vertex is without predecessor in W. If the ordering is close to a topological ordering, however, it is likely that this vertex, v will not have predecessors in W. Even if v does have a predecessor $u \in W$, it is possible that propagation from u will not reach v,

We modify the graph propagator to locally correct the ordering. As we propagate from a vertex to its direct successors, we compare the order number of the vertex to the order number of the successor. If the order numbers do not reflect the order imposed by the dependency graph edge over which we are propagating, we interchange them.

8.2.2 Initial Order Assignment

When we are given a dependency graph D that is to be incrementally evaluated, we first use a (non-incremental) graph evaluator to make D consistent. This evaluator visits all vertices in D in some topological order, and evaluates the vertex function of each vertex in that order.

As we perform this evaluation, we assign an order number to each vertex. We compute this order number as follows. Say that there are approximately n vertices in D, and we have allocated space for a k bit integer ($k \geq \log_2 n$) at each vertex. We put the real (dense) topological order of the vertex in the first $\lceil \log_2 n \rceil$ bits of this integer, and set the remaining $k - \lceil \log_2 n \rceil$ bits randomly. As a special case, if we know that indegree 0 vertices in D are unlikely to get any incoming edges in the future, we assign them very low order numbers (0 for example). Likewise, if we know that certain vertices will never get any outgoing edges, we assign them very high order numbers.

We choose the order numbers this way for several reasons. First, we want the numbering to be sparse to prevent large numbers of vertices from having the same numbers. In the next section we show how this allows us to deal with vertices that have identical order numbers. Second, we want the order numbers for new graph regions to range over the same interval as the order numbers already in D. This gives us an even probability that an arbitrary vertex of the new region will be less than an arbitrary vertex already in D.

8.2.3 Dependency Graph Modifications

If a new component is added to the dependency graph with the *add component* graph modification step (Section 2.3), order numbers must be assigned to the new component. We assign these order numbers in a way similar to the method we used for a complete dependency graph. We perform a topological sort of the the new component and assign k bit order numbers. The number of bits allocated to the real topological order is $\lceil \log_2 n \rceil$ where n is the number of vertices in the new subgraph, rather than the number of vertices in D.

With other graph modification steps, we do not attempt to make the order numbers represent a true topological ordering. It can be helpful, however, to do a constant amount of reordering around the modification points of D. For each edge added from vertex a to v, where we had previously removed an edge from a to b, we give v the order number of b. This gives v a better chance of reflecting the true topological order with respect to its neighbors that are in W. Experimentally, it has improved the performance of the heuristic in the implemented attribute grammar system.

8.2.4 Order Correction

As we propagate from vertex v to vertex u, we locally correct the order numbers by calling the procedure *fix_order_numbers* shown in Algorithm 8.2.1. If the order numbers are the same, we correct the order number for u by adding a random number up to half its bit length. If the order number of v is greater than the order number of u, we simply swap the order numbers.

```
fix_order_numbers(v, u : vertex)
    if order(v) = order(u) then
        number_bits ← ⌈log2order(v)⌉/2;
        order(u) ← order(v) + random(number_bits) + 1
    else if order(v) > order(u) then
        temp ← order(v);
        order(v) ← order(u);
        order(u) ← temp
```

Algorithm 8.2.1

It is unclear if additional order correction is worth the effort. Note, however, that the order correction could be done at any time. If, at any time, a processor is idle, it might be put to work correcting the order numbers of D. For interactive applications, such a process could run between user commands when the processor is otherwise idle.

8.2.5 Graph Propagation with Approximate Topological Ordering

As mentioned in the previous sections, we use the order numbers to approximate the vertex finding process of the *optimal_graph_propagate* procedure of Algorithm 3.3.1. The elements of W are kept in a priority queue. To obtain the vertex v that is first in approximate topological order, we perform a *delete_min* operation on this queue. When we extend propagation to each successor u of v, we call *fix_order_numbers* with v and u as arguments. The approximate topological order propagate procedure, *ato_graph_propagate* is shown in Algorithm 8.2.2.

```
ato_propagate_graph(D : dependency graph, W : vertex priority queue)
    while W ≠ ∅ do
        v ← delete_min(W);
        newvalue ← f_v(values of pred(v));
        if val_v ≠ newvalue then
            val_v ← newvalue;
            for each u ∈ succ(v) do
                fix_order_numbers(v, u);
                W ← W ∪ {u}
```

Algorithm 8.2.2

8.2.6 Performance

Since a sequence of graph modification steps could arbitrarily change the order of the vertices in the dependency graph, we cannot make any statement about how accurate the approximate topological ordering is after a dependency graph modification. Therefore, the worst case running time of *ato_graph_propagate* is the same as that of *naive_propagate_graph* procedure of Algorithm 3.4.1. This running time has been shown to be exponential in the size of DEPENDENT [Reps 84].

If the structure of the existing portion of the dependency graph does not change much during each dependency graph modification, the approximate ordering will be very close to a true topological ordering. In this case, the algorithm performs well. While it is a simple task to devise pathological dependency graphs and dependency graph modifications which cause the dependency graph structure to change in an arbitrary fashion, most meaningful incremental computation seem to exhibit some amount of stability. This area needs more study.

8.3 Concurrent Dependency Graph Modifications

In this thesis, we have assumed that dependency graph modifications begin when the dependency graph is consistent, and that all graph modification steps that make up the graph modification are done before an incremental evaluation is performed.

It is sometimes advantageous to break the strict alternation of dependency graph modifications and incremental evaluations. One might want incremental evaluation to begin as soon as a single dependency graph modification step is made. Also, it might not be necessary for a result to be computed after a dependency graph modification. In this case, we might begin applying the steps of the next modification to the dependency graph. We refer to dependency graph modifications that are not strictly between incremental updates, as *concurrent*.

An example of concurrent dependency graph modifications are the key tree modifications necessary to maintain dynamic collections (Section 7.7). Since the keys needed to determine dependency graph changes were not available until incremental evaluation time, it was necessary to perform dependency graph modifications during graph propagation.

The approximate topological ordering algorithms can be used for concurrent dependency graph modifications. When concurrent modifications are made to the dependency graph, the vertices that are potentially inconsistent are added to the possibly inconsistent vertex set. Say propagation is being performed at vertex v. If the newly modified vertices are beyond v in approximate topological order, they will be updated as though they had been modified before propagation began. Vertices before v in approximate topological order will be evaluated immediately after v has been evaluated.

8.4 Parallel Propagation

With the recent proliferation of parallel shared memory computers and workstations, one must ask what parallelism can be exploited in incremental evaluation. In Section 8.2.4, we mentioned that a background process could be used to correct the approximate topological order numbers when the computer would otherwise be idle. This could be done in parallel as well. In addition, we can also perform propagation in parallel.

While any worklist evaluation algorithm (including topological sort) can be turned into a parallel algorithm by allowing any of a number of processors to remove items from the worklist, approximate topological ordering allows us to do this with very little overhead. This is important in graph evaluation, since the evaluation time of vertex functions is typically short. If the overhead required for a processor to choose a vertex out of the worklist is substantial, each processor will execute at a fraction of the speed of a single processor.

8.4.1 The Parallel Propagation Algorithm

The worklist for the approximate topological order algorithm is a priority queue, W. We convert the *ato_propagate_graph* procedure (Algorithm 8.2.2) into a parallel algorithm over p processes, $P_1, P_2, \ldots, P_p$, by having each process remove vertices from W, which is shared among all processes. As in the single process case, P_i evaluates the function for the removed vertex v, and compares the result with val_v. If the values are different, P_i updates val_v and enqueues the successors of v.

All vertices in the priority queue, however, are not ready for evaluation. It is likely that some of these vertices depend upon other vertices in the priority queue. Since the order of the priority queue approximates a topological order of the dependency graph, vertices deeper in the priority queue are more likely to be dependent than vertices at the head of the priority queue. If propagation reaches the dependent vertex, it will be recalculated. If propagation does not reach the dependent vertex, however, early evaluation of its value will speed the computation. Therefore, we limit the number of processes p to the number of processors that would otherwise be idle. Multiple evaluations of a vertex use an otherwise wasted computing resource.

In the single process case, at most a single vertex evaluation occurs at any given time. In the parallel case, however, many vertices will undergo evaluation at once. It is possible for v's vertex function to be evaluated multiple times in parallel. This can happen as follows. Say that vertex u, one of v's predecessors, changes value while P_i is evaluating v. This can easily happen since the order in which the vertices are being evaluated is approximate. The process that evaluated u will enqueue v. Another process, P_j, is then free to remove v from the priority queue and begin evaluation.

Since we do not assume anything about the relative speed of the processes, P_j could finish evaluating the vertex function before P_i. While P_j would update val_v to have the correct value, P_i would proceed to overwrite val_v with a possibly incorrect value. Even if P_i finishes first, there is no need to enqueue the successors of v until P_j finishes. If evaluation of these successors begin before P_j updates val_v, they will require yet another evaluation.

We avoid such race conditions by using the following locking mechanism. Two additional fields are stored at each vertex v, *vertex_process*(v), and *vertex_enqueued*(v). The first field stores the number of the last process to begin evaluation of the vertex. The *vertex_enqueued* field is a boolean value that indicates whether or not the vertex is in the priority queue. Before placing a vertex v in the priority queue, P_i sets the *vertex_enqueued*(v) field to true.

Upon the removal of a vertex v from the priority queue, P_i sets *vertex_process*(v) to i, and *vertex_enqueued*(v) to false. It then evaluates the vertex function for v, and compares the resulting value to val_v. If the values are not equal, the old value needs to be updated. We must, however, insure that updates to the vertex value are not concurrent. This is accomplished as follows.

First, process P_i locks v, blocking any other process that writes to any of the fields of v. Then, P_i checks the *vertex_process* and *vertex_enqueued* fields. If the vertex is enqueued, it will be reevaluated later when a process dequeues it. If the *vertex_process* field contains the id number of another process, v is being or has been evaluated by another process since it was dequeued by P_i. In both cases, f_v is computed with more up to date arguments, and there is no need to update the vertex value. P_i simply unlocks u and discards the computed value for the vertex function. Otherwise, P_i updates val_v, unlocks v, and enqueues all successors of v that are not already enqueued.

While several processes may be evaluating v at a given time, there can be no interference between them. P_i writes the vertex value of v if v is locked and only if P_i was the last process that dequeued v. Since P_j needs to write to a field of vertex v to begin evaluation, P_j must wait until val_v has been updated and the lock released. Therefore, only the last of the possibly many processes that are evaluating v at a given instant can ever update the vertex value.

Since many vertices are evaluated at a given time, it is possible that vertex v observes the vertex value of vertex u when this value is in an inconsistent state. Because v references val_u, v must be a successor of u. Since all successors of u will be enqueued after u is updated, v will be reevaluated.

Each process continues the evaluation loop as long as the priority queue contains vertices. If the priority queue is empty, the process waits until it is not empty, or until all processes have indicated that they are not busy. If no process is busy and the priority queue is empty, no more vertices will be added and the dependency graph is consistent. This algorithm is shown in Algorithm 8.4.1

8.4.2 Parallel Priority Queue

Note that each vertex in the INFLUENCED set is inserted into and deleted from the priority queue at least once during propagation. Since the priority queue schedules all evaluations for all processes, the *insert* and *delete_min* operations must be inexpensive.

There are two approaches to maintaining a shared priority queue. The first is to use locking to give a single process exclusive access to a conventional priority queue data structure. If the propagation work list remains small, this technique will be successful. If the size of the priority queue becomes large, however, this could become a significant performance bottleneck. If this is the case, we resort to a hardware solution.

We achieve the required performance through the use of a systolic priority queue [Leiserson 81]. While it takes a single processor log n steps to insert an element into

```
ato_propagate_graph_Pi(D : dependency graph, W : vertex priority queue)
    while ∃j.busy[j] = true do
        while W ≠ ∅ do
            busy[i] ← true;
            v ← delete_min(W);
            vertex_process(v) ← i;
            vertex_enqueued(v) ← false;
            newvalue ← f_v(values of pred(v));
            if val_v ≠ newvalue then
                lock v;
                if ¬vertex_enqueued(v)
                and vertex_process(v) = i then
                    val_v ← newvalue;
                    unlock v;
                    for each u ∈ succ(v) do
                        fix_order_numbers(v, u);
                        if ¬vertex_enqueued(u) then
                            vertex_enqueued(u) ← true;
                            insert(W, u)
                else unlock v;
        busy[i] ← false
```

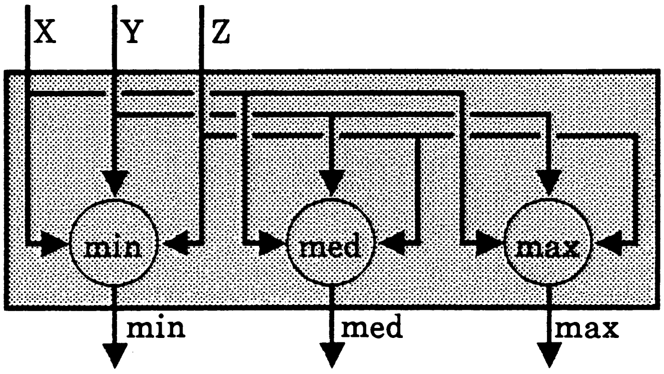
Algorithm 8.4.1

or delete the minimum element of a tree based priority queue of n elements, n processors can perform this task in O(1) time.

The systolic priority queue consists of a chain of n identical processors, each of which sorts three elements. Figure 8.4.1 illustrates this device. The three inputs are labeled X, Y, and Z. The three outputs produce min(X, Y, Z), med(X, Y, Z), and max(X, Y, Z).

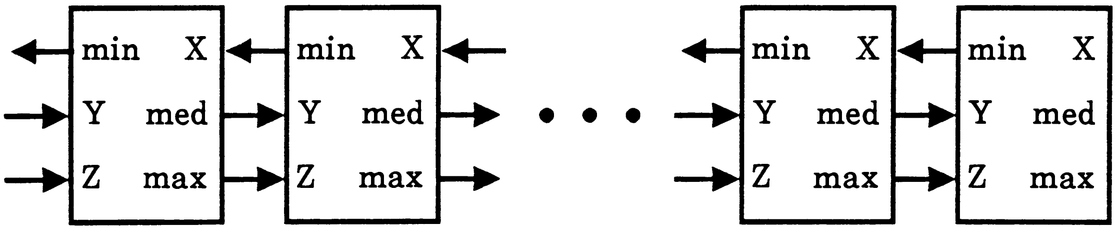
The three-sorters are connected in a chain to make the systolic priority queue. The minimum value is sent to the head of the chain, the medium and maximum values are sent toward the tail of the chain. The chain is shown in Figure 8.4.2.

Each processor in the systolic priority queue works every other clock tick. The details of its operation can be found in [Leiserson 81]. We note, however, that both *insert* operations and *delete_min* operations can be performed every other clock cycle.



Three-Sorter

Figure 8.4.1



Systolic Priority Queue

Figure 8.4.2

We envision many of these processor units fabricated on a single VLSI circuit. These devices would then be cascaded to give the parallel machine an adequately sized priority queue. In addition to storing the priority value, the priority queue would hold a pointer to the memory location representing the corresponding vertex. The priority queue would be attached to the processor bus, allowing each processor to insert a vertex or remove the minimum vertex in a small constant number of bus cycles.

8.4.3 The Amount of Parallelism in Dependency Graphs

It should be emphasized that the parallel propagation algorithm will perform more vertex evaluations than the sequential algorithm. This is because we are evaluating enqueued vertices without waiting for vertices, earlier in approximate topological order, to finish evaluation. If the dependency graph has little parallelism, like the leftmost graph in Figure 8.4.1, little overall speedup can be achieved by the parallel algorithm. If the vertices of this graph were entirely in AFFECTED, almost all computation done in parallel would later be recomputed.

The graph on the right in Figure 8.4.3, however, exhibits a substantial amount of parallelism. A change in the value of v would place the three successors of v into the priority queue. Three processors could then evaluate these vertices in parallel. A change in the value of the first successor would cause three additional parallel evaluations.

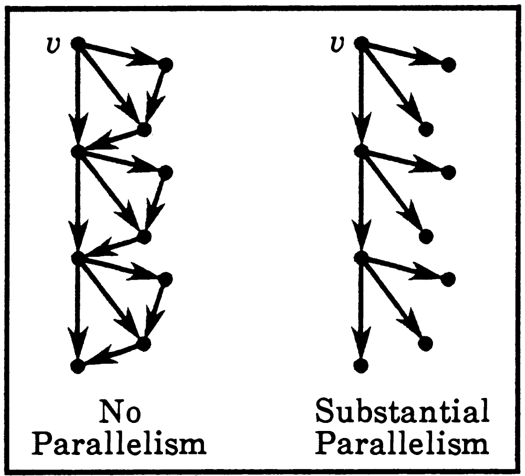


Figure 8.4.3

We assume that the parallel algorithm is executed on a machine whose processors would otherwise be idle. For parallel workstations, this is likely to be the case. If the dependency graph at hand does not contain any parallelism, it will execute in approximately the same time as the sequential algorithm. Dependency graphs with a substantial amount of independent computation, however, will experience a speed up in evaluation of up to the number of processors.

While it is unclear how much parallelism one can expect in dependency graphs, simulations using dependency graphs specified by attribute grammars indicate large amounts of parallelism with as many as 50 processors doing useful work.

Much of the parallel work, however, is caused by the large number of unaffected vertices that are evaluated in aggregates (see Section 7.1). If the unaffected use sites that are dependent upon the aggregate are eliminated from the dependency graph [Hoover and Teitelbaum 86], the speedup achieved is an order of a magnitude smaller. This difference is due to the fact that, in that implementation of aggregates, all definition sites in the aggregate that come after a changed definition must be reevaluated. Since the definition site evaluations are relatively expensive and must be done sequentially, a substantial amount of time is spent with a single processor busy.

We expect that the techniques of Chapter 7 will remove nearly all sequential dependencies from aggregates. Additional research is required to determine how much parallelism can be expected in a given dependency graph and how this parallelism can be increased by transforming the dependency graph.

9. Summary

9.1 Introduction

We propose that incremental graph evaluation be used for incremental computation. By saving the values of intermediate computations at vertices in a dependency graph, the work required to obtain a result after a modification to the computation can be quite small in relation to the amount of work required to perform the entire computation. This reduced computation time will facilitate computer applications that require instant results after minor perturbations in the computation.

9.2 Graph Evaluation

We define the dependency graph, dependency graph evaluation, dependency graph modification, and incremental graph evaluation. A dependency graph represents intermediate computations with vertices and connects the vertex v to vertex u with a directed edge if the associated computation of u depends upon the computation of v. Dependency graph evaluation is the process of computing the desired results from a dependency graph, typically leaving, at each vertex v, a value that is consistent with the values of v's predecessors. If vertices, the computations performed at vertices, and edges between vertices are altered, a set of vertices in the dependency graph become potentially inconsistent. Incremental graph evaluation transforms a dependency graph and a dependency graph modification into a consistent dependency graph.

Graph evaluable schemes, such as attribute grammars and spreadsheets, perform the translation from the application domain to a dependency graph and dependency graph modifications. The fundamental question left unanswered by this thesis is which computer applications can be transformed into dependency graphs by graph evaluable schemes and, given a non-incremental algorithm for such an application, how can a graph evaluable scheme be devised to perform this translation. An answer to this question, together with the results of this thesis, will open quick incremental computation to many application areas.

9.3 Graph Propagation

Graph propagation is a simple technique for transforming a dependency graph and a set of possibly inconsistent vertices into a consistent dependency graph. If graph propagation is to be done efficiently, however, the computations done at each vertex should be performed in some topological order to avoid multiple reevaluations of the same vertex computation. Ordering the vertex evaluations in a topological order, without visiting large portions of the dependency graph, is nontrivial.

We develop a lower bound of O(|INFLUENCED| + EVAL(INFLUENCED)) for graph propagation, where INFLUENCED is the set of vertices that are either directly dependent upon the graph modification or depend upon intermediate values that are changed by the modification, and EVAL(INFLUENCED) is the time required to perform the computation at these vertices. We then summarize methods that have been used to obtain the topological ordering of vertex evaluations for a number of subclasses of dependency graphs and dependency graph modifications.

While some of the ordering methods discussed, such as the algorithm of [Reps 84], obtain the lower bound for some subclasses of dependency graphs and graph modifications, it is not known if all dependency graphs with arbitrary graph modifications can be updated in O(|INFLUENCED| + EVAL(INFLUENCED)) time. More research is needed to determine a tight bound on this problem. If the worst case bound is greater than O(|INFLUENCED| + EVAL(INFLUENCED)) in the general case, what is the largest subclass of dependency graphs and dependency graph modifications that can be updated within the lower bound?

We also introduce several new propagation techniques. By topologically sorting the set of vertices dependent upon the modification, we can update the dependency graph in O(d + EVAL(DEPENDENT)) where DEPENDENT is the set of dependent vertices and d is the size of the dependent portion of the graph. Maintained topological ordering, another ordering technique, allows a true topological order to be incrementally maintained. Approximate topological ordering, discussed in detail in Chapter 8, is a heuristic approximation of a true topological ordering.

9.4 Improving Incremental Graph Evaluation

Even if an optimal graph propagation algorithm is used, the speed of incremental graph evaluation can be improved. The assumptions of the graph propagation model seem to imply that all affected vertices in the dependency graph must be updated. This is not the case. If we construct an improved dependency graph that generates the same results as the original dependency graph, we can eliminate unnecessary computation.

We examine the assumptions that have been made about incremental graph evaluation, and discuss truly optimal incremental graph evaluation. Like the automatic computation of optimal algorithms, such an optimal evaluator is unrealizable. Therefore, we must make more realistic assumptions about graph evaluation. While we do not attempt to optimize the computation at each vertex or change the partitioning of intermediate results, we allow the graph evaluator to examine the computations at each vertex. This examination permits the incremental graph evaluator to construct an improved dependency graph from the original dependency graph such that graph propagation over the improved graph can be performed with a smaller INFLUENCED set.

We construct the improved graph by locating implicit identity dependencies, transitive dependencies in the original graph that communicate values between distant vertices. While, in general, the identification of implicit identity dependencies is a difficult task, the use of some simple primitive operations in the vertex computations allow the evaluator to easily identify many implicit identity dependencies. Since implicit identity dependencies are commonplace and unavoidable in many graph evaluable schemes, propagation on the resulting dependency graph leads to a significant performance improvement.

A major portion of this thesis concerns two classes of implicit identity dependency graphs, copy rule chains and aggregates. Additional research is needed to identify other classes of implicit identity dependencies that are important to incremental computation.

9.5 Structure Trees

Since implicit identity dependencies frequently communicate a value from vertex v, through a tree of dependencies, to a set of designated dependent vertices V', a method is needed to efficiently broadcast a value to V'. We develop the structure tree data structure and algorithms that allow the incremental maintenance of a tree containing a subset of the dependent vertices. This tree can be traversed in $O(|V'|)$ time, allowing the communication in time proportional to the amount required by edges directly bypassing implicit identity dependencies.

The space and time costs of incrementally maintaining structure trees are small. The amount of space required is a small factor of the space required by the dependency graph. A change to a structure tree requires time roughly proportional to the distance in the dependency graph from the modification to the previous designated vertex.

We implicitly make the assumption that structure tree modifications are done on-line. Some sequences of structure tree modifications could be performed with less work if they were done by an off-line algorithm. Additional investigation is required.

9.6 Copy Rule Chains

Copy rule chains are implicit identity dependencies formed by a chain of many vertices, each of which copies the value of the previous vertex. Since in many graph evaluable schemes copy rule chains are necessary to broadcast values, it is important that incremental computation be performed efficiently over copy rule chains in the dependency graph.

We directly apply the structure tree algorithms to accelerate this communication in the dependency graph. In addition, we show how an improved dependency graph D' can be constructed and that D' is optimal for propagation with respect to copy rule chains.

9.7 Aggregates

Aggregates are used to communicate values between keyed definition and use sites in the dependency graph. Many defined values are added to an aggregate value which is communicated throughout a portion of the dependency graph. Use sites then query the aggregate value for the definition with a given key.

If the aggregate value is implemented as a set of definitions at vertices in the dependency graph, all use sites will be dependent upon one of these sets. When a definition site is either added, changed, or removed, all dependent aggregate values will typically change. The INFLUENCED set, therefore, will contain all dependent use sites, even if the use site is unaffected by the altered definition. Graph propagation over this dependency graph will require a large amount of work to restore consistency.

Aggregates represent the communication of values from definition sites to use sites. Replacing these implicit identity dependencies by direct communication between definition and use sites results in a smaller INFLUENCED set. Propagation time is reduced considerably.

We define a collection to be an aggregate constructed with operators that allow empty aggregates to be created, definitions added to the aggregate, and definitions looked up in the aggregate. The operators are used to specify the communication of a set of keyed values through the dependency graph. By examining the vertices of the graph, the implicit identity dependencies between definitions and uses can be determined.

We extend the structure tree data structure and algorithms to allow structure trees to represent the communication between a definition site and its use sites. The structure trees are used to replace the aggregate dependencies in the dependency graph. Graph propagation is performed on the resulting dependency graph. Given a dependency graph modification that affects a definition, the INFLUENCED set for the resulting dependency graph is substantially smaller than the INFLUENCED set for the original dependency graph.

We develop algorithms for incrementally maintaining structure trees for static collections, collections that allow the key at the definition or use site to be determined from information at the vertex, and then extend these techniques for dynamic collections and nested collections. The keys of dynamic collections can be computed from other intermediate values in the dependency graph. Nested collections allow two collections to be combined, resulting in a collection with definitions from both.

Given a change to the collection in the dependency graph, we perform a corresponding change to the structure trees that represent the communication within the collection. The cost of adding or removing a definition or use site to a collection is proportional to the distance in the dependency graph from the modification point to the previous use of the given key. Removing an edge to split a collection, or adding an edge to join two collections roughly corresponds to work proportional to the sum of the distances in the dependency graph from the modification point to the previous reference of each key used beyond the modification point.

The major cost of explicitly representing the definition to use site dependencies is the amount of space required to store on-path sets, sets of keys which have definition-use dependencies that bypass a given vertex in the dependency graph. While the rest of the extended structure tree data structures can be stored in space proportional to the dependency graph, the on-path sets can use as much as $O(k)$ space at each aggregate vertex, where k is the number of keys used in the aggregate. We develop techniques for sharing these on-path sets among vertices. Future research should be directed toward characterizing the amount of sharing possible and determining a lower bound on the amount of space required to represent these sets.

Dynamic collections also present difficulties. Since the aggregate keys are computed by the dependency graph, the keys can change during graph propagation. This corresponds to a change in the dependency graph during propagation. If only a few keys change with every incremental evaluation, these dependency graph modifications are manageable. Techniques to characterize the number of key changes in a given aggregate are needed.

We limit the computation that comprises the aggregate to a small set of primitive operations, restricting the structure of the aggregate to be explicit in the dependency graph. While it appears to be possible to eliminate these restrictions, it is not clear that explicitly representing the resulting dependencies would result in an improved incremental evaluation time.

9.8 Approximate Topological Ordering

Approximate topological ordering is a graph propagation algorithm that uses a heuristic approximation of a topological order of the dependency graph. While in the worst case this algorithm can perform as bad as graph propagation using any ordering of the dependency graph, it has performed well in practice. An implementation using the dependency graphs generated by attribute grammars with subtree replacement modifications typically performs a number of unnecessary evaluations that is a few percent of the total number of evaluations.

We give extensions to the approximate topological ordering algorithm to allow concurrent dependency graph modifications, modifications that occur during propagation, and to allow parallel propagation using a shared memory parallel computer.

While approximate topological ordering is no better than any other ordering in the worst case on arbitrary dependency graphs for any graph modification, research is needed to determine if it has better worst case or expected time performance on a subset of dependency graphs and dependency graph modifications. Experimental implementations are also needed for other problem domains to determine the algorithm's practical importance.

Simulations of the parallel algorithm have shown a large speedup if the dependency graph has a large amount of inherent parallelism, and very little speedup if the computations are mostly sequential. The collection algorithms presented in this thesis should enhance parallelism in dependency graphs with aggregates. Other sources of sequential dependencies in dependency graphs should be identified and corrected if possible. An implementation on a parallel machine would provide much insight.

9.9 Summary

Dependency graphs can be used to represent incremental computation. Incremental graph evaluation is the process of computing the new results of a computation after the dependency graph has been altered. We use a graph propagation algorithm to restore consistency to a modified dependency graph. Improving communication in the dependency graph increases the speed of graph propagation. We can incrementally maintain data structures that explicitly represent the communication of values between dependency graph vertices to overcome restrictions inherent in many dependency graphs. The result is a practical improvement in the speed of incremental graph evaluation.

Bibliography

[Aho, Hopcroft, and Ullman 74]
Aho, A., Hopcroft, J., and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[Barford 87]
Barford, L. A Graphical, Language-Based Editor for Generic Solid Models Represented by Constraints, Ph.D. Thesis, Cornell University, May 1987.

[Bates and Constable 85]
Bates, J. and R. Constable. Proofs as Programs. *ACM Transactions on Programming Languages and Systems* 7, 1 (January 1985), pp. 113-136.

[Beshers and Campbell 85]
Beshers, G., and R. Campbell. Maintained and Constructor Attributes. *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 1985, pp. 34-42.

[Brown and Tarjan 79]
Brown, M. and R. Tarjan. A Fast Merging Algorithm. *Journal of the ACM* 26, 2 (April 1979), pp. 211-226.

[Chamberlin, King, Slutz, Todd, and Wade 81]
Chamberlin, D., J. King, D. Slutz, S. Todd, B. Wade. JANUS: An Interactive System for Document Composition. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 1981, pp. 82-91.

[Cohen and Harry 79]
Cohen R. and E. Harry. Automatic Generation of Near-Optimal Linear-Time Translators for Non-Circular Attribute Grammars. *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1979, pp. 121-134.

[Demers, Reps, and Teitelbaum 81]
Demers A., Reps T., and T. Teitelbaum. Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors. *Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Williamsburg, Virginia, January 1981, pp. 105-116.

[Demers, Rogers, and Zadeck 85]
Demers, A., A. Rogers, and F. Zadeck. Attribute Propagation by Message Passing. *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 1985, pp. 43-59.

[Dietz and Sleator 87]
Dietz, P. and D. Sleator. Two Algorithms for Maintaining Order in a List. To appear at the 19th Annual ACM Symposium on Theory of Computing, New York, New York, May 1987.

[Driscoll, Sarnak, Sleator, and Tarjan 86]
Driscoll, J., N. Sarnak, D. Sleator, and R. Tarjan. Making Data Structures Persistent. *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, Berkeley, California, May 1986, pp. 109-121.

[Farrow 86]
Farrow, R. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 1986, pp. 85-98.

[Hoover 86]
Hoover, R. Dynamically Bypassing Copy Rule Chains in Attribute Grammars. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1986, pp. 14-25.

[Hoover and Teitelbaum 86]
Hoover, R. and T. Teitelbaum. Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 1986, pp. 39-50.

[Hudson 86]
Hudson, S. A User Interface Management System Which Supports Direct Manipulation, Ph.D. Thesis, University of Colorado, August 1986.

[Jalili 85]
Jalili, F. A General Incremental Evaluator for Attribute Grammars. *Science of Computer Programming*, 5 (1985), pp. 83-96.

[Johnson 84]
Johnson, G. An Approach to Incremental Semantics. Ph.D. Thesis (TR 547), University of Wisconsin, Madison, July 1984.

[Johnson and Fischer 82]
Johnson, G. and C. Fischer. Non-syntactic Attribute Flow in Language Based Editors. *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1982, pp. 185-195.

[Johnson and Fischer 85]
Johnson, G. and C. Fischer. A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 1985, pp. 141-151.

[Jones and Simon 86]
Jones, L. and J. Simon. Hierarchical VLSI Design Systems Based on Attribute Grammars. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1986, pp. 58-69.

[Kastens 80]
Kastens, U. Ordered Attribute Grammars. *Acta Informatica* 13, 3 (1980), pp. 229-256.

[Kastens, Hutt, and Zimmermann 82]
Kastens, U., B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*. Lecture Notes in Computer Science, Springer-Verlag, Berlin/Heidelberg/New York, 1982.

[Kaplan 86]
Kaplan, S. Incremental Attribute Evaluation on Graphs. Technical Report UIUC-DCS-86-1309, University of Illinois at Urbana-Champaign, December 1986.

[Kay 84]
Kay, A. Computer Software. *Scientific American* 251, 3 (September 1984), pp. 53-59.

[Kennedy and Ramanathan 79]
Kennedy, K and J Ramanathan. A Deterministic Attribute Grammar Evaluator based on Dynamic Sequencing. *ACM Transactions on Programming Languages and Systems* 1, 1 (July 1979), pp. 142-160.

[Kennedy and Warren 76]
Kennedy, K. and S. Warren. Automatic Generation of Efficient Evaluators for Attribute Grammars. *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, Atlanta, Georgia, January 1976, pp. 32-49.

[Knuth 68]
Knuth, D. Semantics of Context-free Languages. *Mathematical Systems Theory* 2,2 (June 1968), pp. 127-145.

[Knuth 73]
Knuth, D. *The Art of Computer Programming Volume 1/ Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1973.

[Leiserson 81]
Leiserson, C. Area-Efficient VLSI Computation. Ph.D. Thesis (TR CMU-CS-82-108), Carnegie-Mellon University, October 1981.

[Nelson 85]
Nelson, G. Juno, a Constraint-Based Graphics System. *Proceedings of the ACM SIGGRAPH '85 Symposium*, San Francisco, California, July 1985, pp. 235-243.

[Reiss 84]
Reiss, S. An Approach to Incremental Compilation. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 144-156.

[Reps 82]
Reps, T. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1982, pp. 169-176.

[Reps 84]
Reps, T. *Generating Language-based Environments.* MIT Press, Cambridge, Massachusetts, 1984.

[Reps 86]
Reps, T. Incremental Evaluation for Attribute Grammars with Unrestricted Movement Between Tree Modifications. Computer Sciences Technical Report #671, University of Wisconsin, Madison, October 1986.

[Reps, Marceau, and Teitelbaum 86]
Reps, T., C. Marceau and T. Teitelbaum. Remote Attribute Updating for Language-Based Editors. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1986, pp. 1-13.

[Reps and Teitelbaum 84]
Reps, T. and T. Teitelbaum. The Synthesizer Generator. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, April 1984, pp. 42-48.

[Reps, Teitelbaum, and Demers 83]
Reps, T., T. Teitelbaum, and A. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems* 5, 3 (July 1983), pp. 449-477.

[Rosen 77]
Rosen, B. High-Level Data Flow Analysis. *Communications of the ACM*, 20, 10 (October 1977), pp. 712-724.

[Ross 85]
Ross, R. Design of Personal Computer Software. *Insights Into Personal Computers*, (Gupta, A. and H. Toong editors). IEEE Press, New York, 1985, pp. 282-300.

[Staudt, Krueger, Habermann, and Ambriola 86]
Staudt, B., C. Krueger, A. Habermann, and V. Ambriola. The GANDALF System Reference Manuals. Technical Report CMU-CS-86-130, Carnegie-Mellon University, May 1986.

[Tarjan 83]
Tarjan, R. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[Teitelbaum and Reps 81]
Teitelbaum, T. and T. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24, 9 (September 1981), pp. 563-573.

[Tsakalidis 84]
Tsakalidis, A. Maintaining Order in a Generalized Linked List. *Acta Informatica* 21 (1984), 101-112.

[Waite and Goos 84]
Waite, W. and G. Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

[Warren 76]
Warren, S. The Coroutine Model of Attribute Grammar Evaluation. Ph.D. Thesis, Rice University, Houston, Texas, April 1976.

[Yeh 83]
Yeh, D. On Incremental Evaluation of Ordered Attributed Grammars. *BIT* 23 (1983), pp. 308-320.