Dynamically Bypassing Copy Rule Chains in Attribute Grammars

Roger Hoover

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

Attribute grammars require copy rules to transfer values between attribute instances distant in an attributed parse tree. We introduce copy bypass attribute propagation that dynamically replaces copy rules with nonlocal dependencies, resulting in faster incremental evaluation. A evaluation strategy is used that approximates a topological ordering of attribute instances. The result is an efficient incremental evaluator that allows multiple subtree replacement on any noncircular attribute grammar.

1.0 Introduction

In a standard attribute grammar, a semantic equation may only refer to attribute values of the same production instance in the corresponding context free grammar. If a semantic equation generates a value that is to be used by an attribute instance distant in the tree, the value must be propagated through semantic equations whose only purpose is to move the value to the next production in the parse tree. These attribute instances are referred to as *copy attributes* and their defining semantic equations are called *copy rules*. The repeated computation of these copy attributes during incremental evaluation is necessary if the value at the beginning of the copy chain changes. The time required for an incremental evaluator to copy these values is substantial.

Instead of propagating attribute values through copy rule chains, we wish to transfer attribute values directly from their creators to their users by means of nonlocal dependencies. Copy bypass attribute propagation dynamically replaces copy rule chains in an attributed grammar with copy bypass (CB) dependencies.

We present algorithms that install CB dependencies, perform change propagation over CB dependencies, and remove CB dependencies that become invalid after a tree modification.

In order to perform incremental evaluation after a modification, we need an incremental evaluator that works in the presence of nonlocal dependencies. We introduce an incremental graph evaluator that has this property. Approximate topological ordering maintains a locally accurate topological order of the graph that is used for evaluation priority.

We have implemented these two strategies in the Synthesizer Generator [RT84]. While a controlled performance evaluation has not yet been done, our initial results are quite promising. When performing incremental evaluation in a syntax directed Pascal editor, copy bypass propagation significantly reduces the time necessary to update symbol table and other far reaching changes. For extended editing sessions, extra evaluations caused by incorrect topological ordering information is negligible, typically averaging around 2% of necessary evaluations. In the worst cases that we have observed, this rate remains well under 50%. The overall performance of this evaluation strategy is similar

This work is sponsered in part by a grant from IBM

to our implementation of the evaluation strategy for ordered attribute grammars [K80, Y83]. Our strategy, however, is not limited to a subclass of attribute grammars. Any attribute grammar, including those with nonlocal dependencies, can be incrementally evaluated.

In Section 2.0 of this paper, we argue that bypassing copy rule chains is desirable and we discuss previous work in the area. In Section 3.0 we introduce copy bypass attribute propagation and give an incremental update algorithm that maintains CB dependencies. We introduce an evaluation scheme in Section 4.0 that allows us to perform change propagation in the presence of nonlocal dependencies. In Section 5.0 we combine copy bypass attribute propagation with approximate topological ordering for a complete incremental update algorithm. Section 6.0 gives two methods of reducing the storage costs of copy bypass propagation. Time and space bounds are discussed in Section 7.0.

2.0 Motivation and Previous Work

Numerous evaluation algorithms have been published for incrementally updating attribute grammars after changes. Some of these algorithms, i.e. [RTD83, R84], can be applied to any noncircular attribute grammar, but others, [Y83, JF85], are restricted to subset classes of attribute grammars. Many of these algorithms are optimal in the sense that they run in time O(|*Affected*|), where *Affected* is the set of attribute instances whose values change with a modification of the attributed tree. This running time is achieved by forcing the evaluation of attributes to be in topological order with respect to the attribute dependency graph.

By eliminating propagation over copy rule chains, we can reduce the size of *Affected*. Since the copy rule portion of *Affected* can be arbitrarily large, CB propagation can perform an incremental update faster than these conventional update algorithms.

Consider the attribute grammar in Figure 1. Copy rules are indicated by $=_c$.

An attributed parse tree for *let a = 5 in a+4+8 ni* is shown in Figure 2. Notice how the symbol table is needlessly copied into the declaration of *a* and how it is copied through the tree in order for

S::=E	T::=let D in E ni
S.val = E.val	T.val $=_c$ E.val
E.st = ∅	D.st $=_c$ T.st
	E.st = update(T.st,D.df)
E::=T	T::=id
E.val $=_c$ T.val	T.val = lookup(T.st,id)
T.st $=_c$ E.st	
	T::=int
E_1::=E_2+T	T.val = valueof(int)
E_1.val = E_2.val	
+T.val	D::=id = E
E_2.st $=_c$ E_1.st	D.df = [id,E.val]
T.st $=_c$ E_1.st	E.st $=_c$ D.st

Figure 1

the value of *a* to be determined. If we were to graft the subtree under the first E nonterminal into another parse tree where st is not empty, a conventional evaluator would update all of these copy attributes. In our implementation, the attributed tree for a 500 line Pascal program to format text files [KP81] contains some 5400 copy chains, approximately half of which are terminated by uses. During incremental evaluation, copy bypass propagation allows us to avoid evaluating copy chains not terminated by uses, and bypass all others.

Efforts have been made to extend the attribute grammar formalism by allowing nonlocal dependencies. In [J84, JF85] attribute grammars are extended to allow nonlocal productions. The dependencies created by these productions, however, must be explicitly specified in the grammar and cannot be used to eliminate all copy rule chains. In [DRZ85], dependencies are established by the transmission of messages between attributes. While these explicitly specified dependencies are established between attribute instances remote in the parse tree, unconditionally forwarded messages are analogous to copy rules and the method requires propagation over them. During incremental evaluation, copy bypass propagation avoids propagation over copy chains by creating implicit nonlocal dependencies.

The introduction of nonlocal dependencies complicates the attribute evaluation process, especially if these dependencies are created dynamically. Although the algorithms of [K80, R84] have been extended for a subclass of

nonlocal dependencies [RMT86], this subclass does not include all nonlocal dependencies needed to bypass copy rule chains. In [JF85] and in [DRZ85], an analysis is made of possible dependencies and attributes are given a priority number that specifies their evaluation order. If this order cannot be statically determined, the attribute grammar cannot be evaluated by the strategy. [JF85] allow the evaluation of some grammars that fail this test if a traversal order of the tree is maintained. [DRZ85] observe that a topological ordering would allow all grammars to be evaluated, but do not give an algorithm for maintaining one. Approximate topological ordering uses a heuristic approximation of the true topological order and can evaluate any noncircular attribute grammar.

3.0 Copy bypass Attribute Propagation

Copy bypass attribute propagation is a method of attribute propagation that allows copy rule chains to be bypassed. Section 3.1 shows how copy bypass (CB) dependencies are dynamically inserted. In Section 3.2 we show how invalid CB dependencies can be removed after a tree modification. The process of reestablishing CB dependencies is discussed in Section 3.3.

3.1 The Insertion of CB Dependencies

Upon the initial attribution of the tree, we give each copy attribute a special value, called a CB value, instead of the value of its semantic equation. A *CB value* is a pair consisting of the *location* where its true value can be found, and a *pathkey*, a string that specifies the path taken from this location. Let a be the last attribute instance in a copy rule chain. When the value a is needed to compute the value of its successor, we create a *CB dependency* by placing the location of a and the pathkey stored at a in a lookup tree at the head of the chain. We order the *CB lookup tree* by pathkey in lexigraphical order.

We define the following functions for attribute instance a. We will use these functions to specify how the tree is attributed.

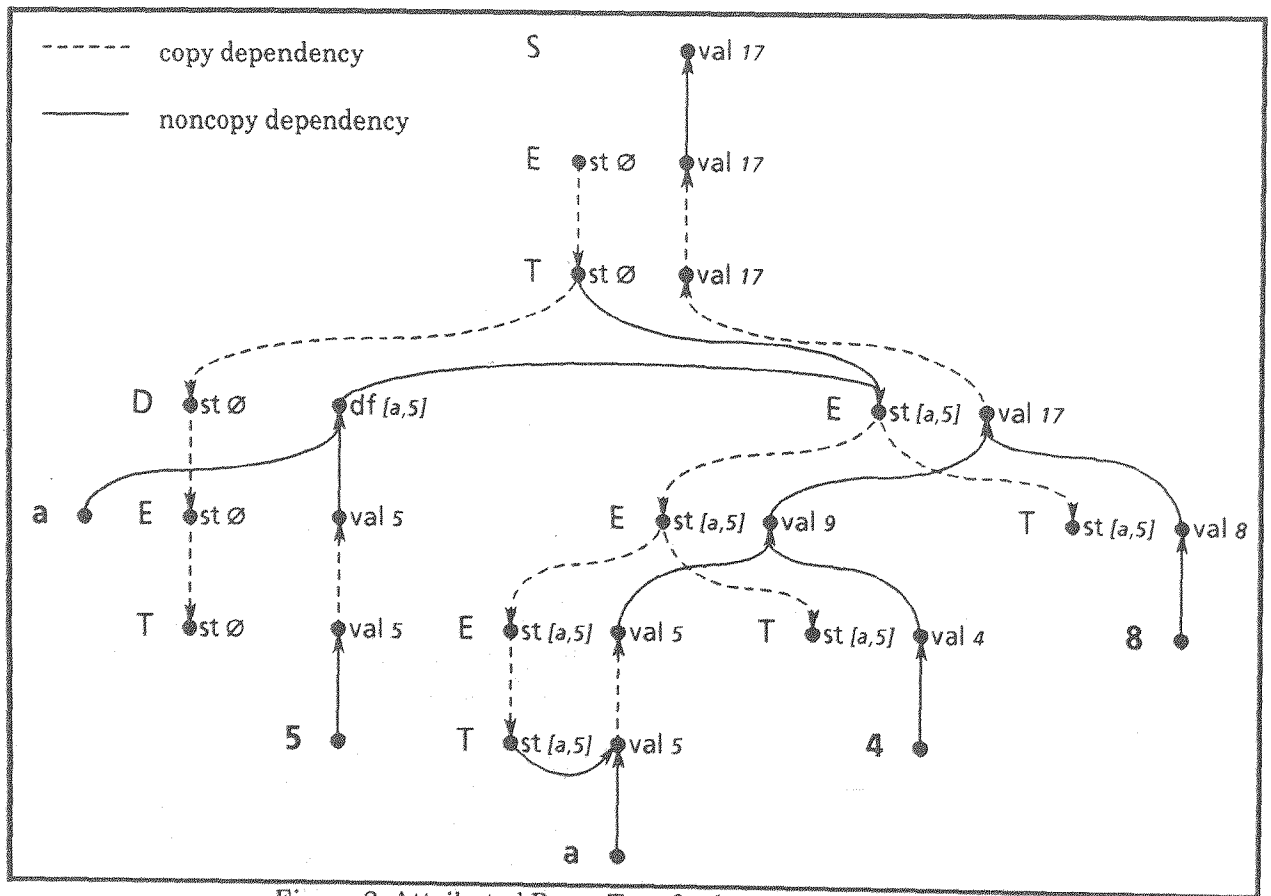

Figure 2: Attributed Parse Tree for *let a=5 in a+4+8 ni*

value(a) The value of attribute instance a. This is the CB value if a is defined by a copy rule.

location(a) The location of attribute instance a. Since this location uniquely identifies a, we will sometimes use the location of an attribute in place of its name.

succ(a) The set of attribute instances that directly depend on attribute instance a. This set does not include any nonlocal dependencies.

$\text{index}_b(a)$ We number the successor set of b from 0 to $|\text{succ}(b)|-1$. This function returns the number of attribute instance a in the successor set of b. Since copy attributes have only one predecessor, we drop the subscript and refer to index(a) for copy attribute instance a.

$\text{concat}_b(k, a)$ Given an attribute instance b that a directly depends on, we concatenate the $\text{index}_b(a)$ onto the end of a pathkey k. We will drop the subscript for copy attribute instance a as we did for the index function. Since we need to delimit the index from the rest of the pathkey, think of concat(k,a) as k||.||index(a).

We give values to attribute instances as follows.

Say the semantic function that defines a is f. If f is a copy rule, we give a a CB value $<loc, path>$. If b, the attribute instance being copied by f, does not have a CB value, loc is the location of b and $path$ is the index number of a appended to the empty string. If b has a CB value $<lb,pb>$, $loc=lb$ and $path$ is computed by concatenating the index number a to the end of pb.

If f is not a copy rule, a will have the value defined by f. If an argument b to f has a CB value $<l,p>$, the value stored at location l is used. A nonlocal dependency is then established from l to b and stored under pathkey p in a lookup tree at l.

In the future, when the value of l changes, we will propagate this change directly to a.

Thus, when computing the value of the attribute, we evaluate the CB semantic function (shown in Figure 3) instead of evaluating f. The function add_CB_dependency(l,b,p) inserts the pair <location(b), p> in the CB lookup tree at attribute location l.

```
function CB_sem_func(a : attribute_instance)
  {say the sem. func. defining a is f(b1, ... , bn)}
  if is_copy_rule(f) then
    if ¬has_CB_value(b1) then
      return(<location(b1), concat(ε, a)>)
    else
      <l, p>←value(b1);
      return(<l, concat(p, a)>)
  else
    return(f(true_val(b1), ..., true_val(bn)))

function true_val(b : attribute_instance)
  if ¬has_CB_value(b) then return(value(b))
  else
    if ¬has_CB_dependency(b) then
      <l, p>←value(b);
      add_CB_dependency(l,b,p);
      has_CB_dependency(b)←true;
    return(true_val(l))
```

Figure 3

To incrementally update a tree that has been attributed using the CB semantic function (Figure 3), we use simple change propagation and propagate over both local and nonlocal dependencies. The procedure propagate, shown in Figure 4, takes the set of all inconsistent attributes and consistently attributes the tree. Note that this procedure requires that we be able to determine the first element in evaluation set in some topological order. In Section 4 we will show how this can be replaced by an approximate topological ordering.

Attributing the parse tree in Figure 2 using the copy bypass semantic function results in the attribute dependency graph shown in Figure 5. In this figure, an integer before the name of an attribute indicates its location. In the actual implementation, the location is the address of the storage cell containing the attribute. Note that all copy chains have been bypassed by CB dependencies. Although the CB dependencies

```
procedure propagate(S : set of attributes)
  while S ≠ ∅ do
    a←first_in_topological_order(S);
    S←S−{a}
    newvalue←CB_sem_func(a);
    if value(a) ≠ newvalue then
      value(a)←newvalue;
      S←S∪succ(a)∪nonlocal_succ(a)

function nonlocal_succ(b : attribute_instance)
  N←∅;
  if has_CB_lookup_tree(b) then
    for all c∈CB_lookup_tree(b) do
      N←N∪succ(c)∪nonlocal_succ(c);
  return(N)
```

Figure 4

point to the last copy attribute in the chain, the evaluator propagates changes directly to their successors. For example, the use of the identifier a depends upon the attribute where the symbol table is modified.

3.2 Modification of the Attributed Tree

We use multiple subtree replacement as our model of tree modification. Given an attributed tree T with subtrees T_1, T_2, ..., T_n, we replace T_1, T_2, ..., T_n with T_1', T_2', ..., T_n' such that the root of T_i has the same nonterminal as the root of T_i'. The nonterminals at the points of subtree replacement are referred to as the *intersection nonterminals* and their attribute instances are referred to as *intersection attributes*. Note that we are not restricted to subtree replacement. In Section 4 we will introduce a more general modification model based upon the division of the attribute dependency graph by a vertex cut set.

As it is done in [RTD83, R84], we replace the inherited intersection attribute values with those from the subtree and keep the synthesized intersection attribute values from T. For attribute grammars in normal form, this confines all inconsistent non copy attributes to the intersection nonterminal. Our change propagation algorithm (Figure 4), however, does

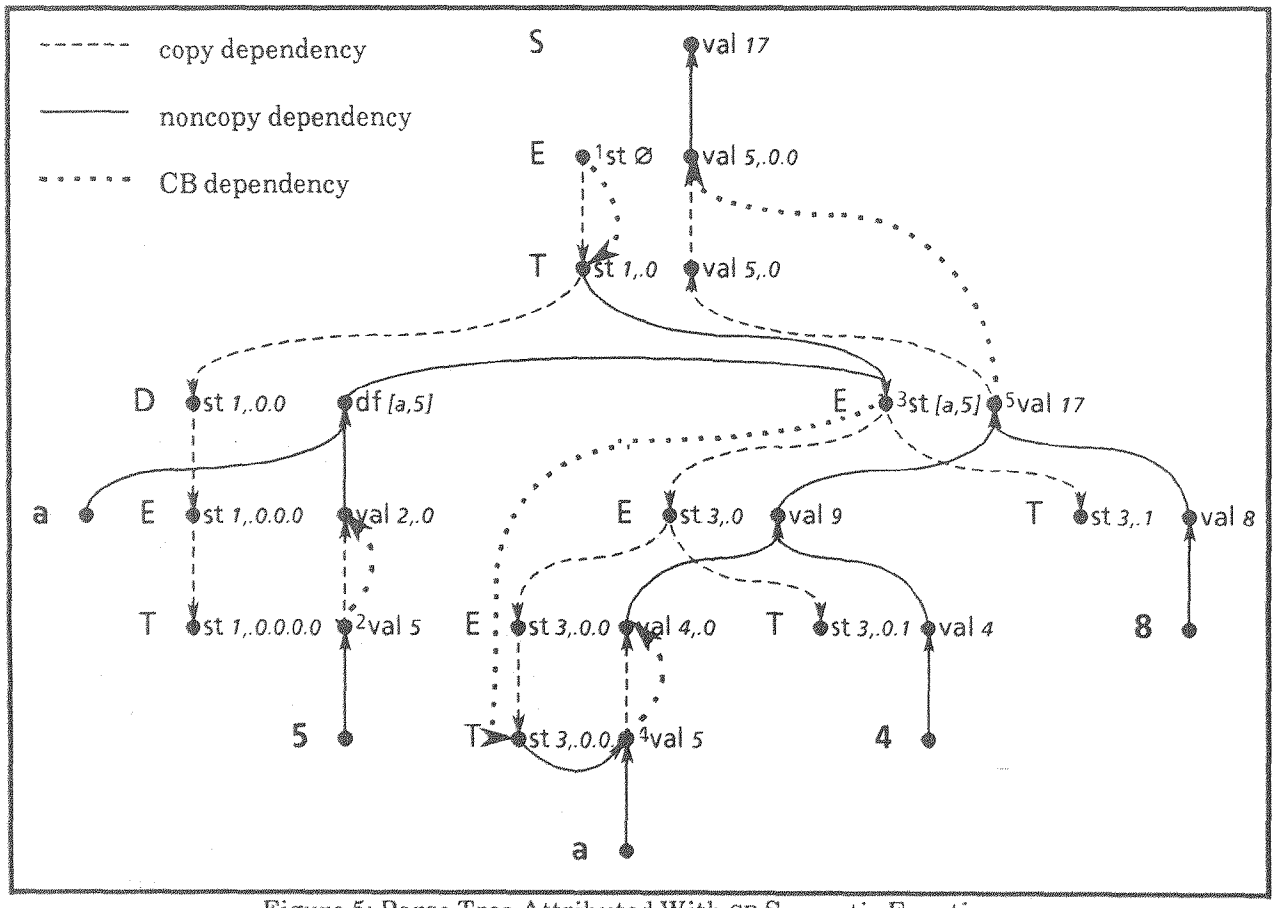

Figure 5: Parse Tree Attributed With CB Semantic Function

not require this. We need only to be able to determine which attribute instances are inconsistent.

We are using location pointers to represent CB dependencies. Thus, the above replacement strategy will not confine all inconsistent copy attributes to the intersection nonterminal if one of these pointers refers to the location of a replaced value. Therefore, we must not only replace the intersection attribute values, but we must insure that the replacing attributes have the same locations as the attributes that they replace. If the implementation does not allow this, all nonlocal dependencies from the replaced attribute are invalid and must be removed. This is done by considering the copy rule successors of such intersection attributes part of the intersection in the following discussion.

Before we can perform the subtree replacement, we must detect and remove any CB dependencies that become invalid when each T_i is separated from T. There are three cases in which a CB dependency might be invalid.

1. The copy rule chain bypassed by the dependency crosses the intersection.

2. The dependency ends at an intersection attribute.

3. The dependency begins at an intersection attribute.

These three cases are shown in Figure 6. Note that these cases are not exclusive.

For each CB dependency of cases 1 and 2, there must exist an intersection attribute x with a CB value. If the inherited attributes come from T_i' and the synthesized attributes come from T, none of these dependencies will be valid after the subtree replacement. Therefore, we must remove them.

Say value(x) = <*loc*,*path*>. Any CB dependencies that were created to bypass x must originate from the attribute at location *loc*. Since we created the pathkeys for these dependencies by concatenating additional path information to the end of *path*, the pathkeys of any CB dependencies that bypass x must have *path* as a prefix. It is easy to see from the construction of the pathkeys

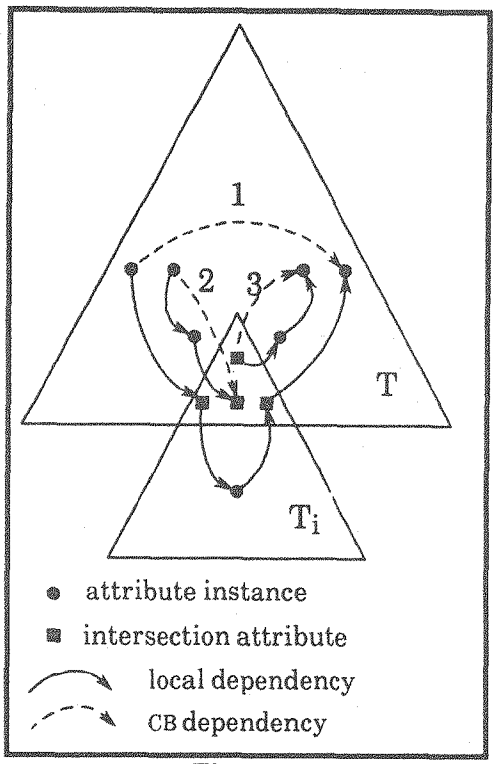

Figure 6

that all other dependencies from *loc* will not have *path* as a prefix. Therefore, we can remove all case 1 and 2 dependencies by deleting all CB dependencies that have pathkeys prefixed by *path* from the CB lookup tree at *loc*. Since the lookup tree is kept in lexigraphical order, we can do this operation by splitting the tree into three trees, $L_1 < path$, $path \le L_2 <$ successor(*path*)†, and $L_3 \ge$ successor(*path*). Joining L_1 and L_3 results in the desired tree with all invalid dependencies bypassing x removed.

Any remaining invalid dependencies must be of case 3 and will have a CB lookup tree at an intersection node x. If the grammar is in normal form, all inherited attributes will have CB successors only in T_i' and they will remain valid. Likewise, all synthesized attributes will have CB successors only in T. If the grammar is not in normal form, we must remove CB dependencies that bypass any successors of x that violate normal form. We do this exactly as we did for the intersection attributes in cases 1 and 2.

†By successor of a string s, we mean the next string in lexigraphical order that does not have s as a prefix or, if no such string exists, a final string greater than all strings in the language.

The algorithm to remove invalid dependencies is shown in Figure 7. Non_normal_form_succ(x) is the set of successors of x defined in the same production as x is defined.

```
procedure remove_invalid_dependencies
                          (T : tree, T_i : subtree)
  for all x∈attr_instances(root(T_i)) do
    if has_CB_value(x) then
      <l, p> ←value(x);
      CB_tree_remove(l, p)
    else if has_CB_lookup_tree(x) then
      for all b∈non_normal_form_succ(x) do
        if has_CB_value(b) then
          <l, p> ←value(b);
          CB_tree_remove(l, p);

procedure CB_tree_remove
            (l : attribute_location, p : pathkey)
  L←CB_lookup_tree(l);
  split_tree(L, p, L1, L2);
  split_tree(L2, successor(p), L2, L3);
  remove_tree(L2);
  CB_lookup_tree(l)←join_tree(L1, L3)
```

Figure 7

3.3 Reestablishing CB Dependencies

The process of reestablishing CB dependencies after a subtree replacement is automatic--it takes place as inconsistent CB values are propagated to consistency by the propagation algorithm. Note that it is possible for change propagation to quiesce before the end of a copy rule chain is reached. This can happen if either a subtree is replaced with itself or if a replaced subtree completes a previously established copy rule chain. To reestablish the necessary CB dependencies, we could force the evaluation of inconsistent copy rule chains to the finish. With some subtree replacements, this would require that we unnecessarily traverse a potentially large set of copy attributes.

We avoid these unnecessary evaluations by placing the destination attribute of all removed CB dependencies into the proper inconsistent attribute set. To be able to do this, we must determine the location of these attributes--either in the main tree or in subtree T_i. This is necessary to determine the initial evaluation set to give the propagate procedure (Figure 4). If the removed subtrees are to be swapped into another tree, we must keep the set of such attributes in the subtree until the replacement is done. When this subtree is used in a subtree replacement, the attribute set is added to the initial evaluation set.

In our implementation, this location is determined by observing the direction, into or out of the subtree, of the copy chains crossing the intersection. A copy chain crossing a multiple number of times is indicated by multiple CB values at the intersection that have the same location pointer.

We form an ordered *removal trees* of these values with respect to the partial order imposed by the prefix relation on the pathkeys. Each totally ordered *removal chain* in a removal tree corresponds to the multiple intersection crossings of a copy chain, each element of the removal chain indicating an intersection crossing. The invalid CB dependencies indicated by the last CB value in this removal chain must be located after the intersection point in the copy chain where this CB value was found.

Thus, we can identify the location of the attributes pointed to by CB dependencies bypassing the last element in the removal chain. This process can be recursively repeated for the rest of the removal chain. Since the removal trees are in prefix order, CB dependencies specified by the CB values at the removal tree root, R, include all dependencies specified by CB values in the rest of the removal tree. Thus, the invalid dependency removal from section 3.2 can be done top down.

The root of the removal tree specifies a subrange to be removed from the dependency tree at the copy chain head. Recursively, each child of R then indicates a subrange to remove from its parent's tree. The resulting dependency tree for each node in the removal tree indicates the CB dependencies to destinations after the corresponding intersection point but before the next intersection point. The locations of the dependency destinations are then determined from the copy chain direction at the intersection point for each subtree.

In our tests, this optimization resulted in a reduction of a few percent in the total number of attributes evaluated. The simple tree modification algorithm is shown in Figure 8.

```
procedure modify_tree(T : tree,
                      S : subtree_pair_set)
  inconsistent←∅;
  for all <subtree, subtree'>∈S do
    remove_invalid_dependencies(T, subtree);
    replace_subtree(T, subtree, subtree');
    for all b∈attr_instances(root(subtree)) do
      inconsistent←inconsistent∪{b}
            ∪non_normal_form_succ(b);
  propagate(inconsistent)
```

Figure 8

4.0 Approximate Topological Ordering

Approximate topological ordering is a graph evaluation strategy that relies upon a heuristic approximation of a topological ordering of the graph. In Section 4.1 we define the topological ordering problem and discuss its application to graph evaluation. Section 4.2 describes the initial order assignment and attribution of the graph. Correction of the assigned order is described in Section 4.3. Section 4.4 gives the incremental graph evaluator.

4.1 Topological Ordering Problem and Graph Evaluation

For a directed acyclic graph G with nodes N and $S \subseteq N$, we define *find_first* to be the operation that finds $b \in S$ such that b comes before all other nodes of S in some topological ordering of G. We may also perform *mod_graph* operations which modify the graph as follows. G is divided by vertex cut set V into G', C and another graph component C' is grafted in the place of C. This modification is illustrated in figure 9. The effect on the attribute dependency graph in an attributed tree by multiple subtree replacement is a special case of such a graph modification.

The topological ordering problem is to efficiently perform repeated *find_first* operations interspersed with an occasional *mod_graph* operation. It is difficult because it requires the quick computation of a global property of the graph--the topological order. A graph modification can drastically change this order.

We wish to evaluate a semantic equation for each node in a dependency graph. Given a solution to the above problem, we could construct an incremental graph evaluator by performing

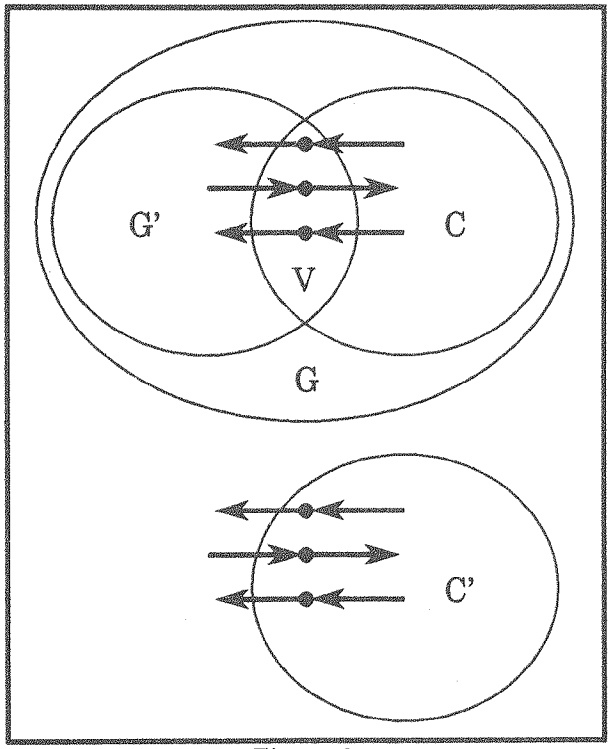

Figure 9

change propagation in topological order. Since we could not evaluate a node before one of its predecessors, we would never evaluate nodes more than once. While we do not have a solution to the topological ordering problem, we do have a heuristic algorithm that approximates it. Our approach is to maintain a locally accurate topological ordering. This exploits the locality property of change propagation. While we may incorrectly order nodes distant in the graph, nodes close to each other in the graph will be ordered correctly.

4.2 Initial Attribution and Order Assignment

We initially evaluate each node in the dependency graph. We do this by performing a topological sort of the graph, evaluating the nodes in that order. As we perform this evaluation, we assign an order number to each node. We want these order numbers to be in topological order, somewhat random, and to be distributed over a range much greater than the number of nodes in G. We do this by assigning nodes with indegree 0 a very low order, and creating order numbers for other nodes by appending random digits to the real topological order. We have implemented this order number as a 32 bit integer, the first $\lceil \log_2 n \rceil$ bits containing the initial topological order, the

rest random. The initial attribution algorithm is shown in Figure 10.

```
procedure initial_attribution(G : graph)
  num_order_bits←32;
  random_bits←num_order_bits − ⌈log2|G|⌉;
  for all node∈G do
    edges_left(node)←indegree(node);
  S←{node∈G | indegree(node) = 0};
  top_order←0;
  while ∃b∈S do
    top_order←top_order + 1;
    if indegree(b) = 0 then order(b)←1
    else order(b)←
          shift_left(top_order, random_bits)
          + random(random_bits);
    value(b)←semantic_function(b);
    for all c∈succ(b) do
      edges_left(c)←edges_left(c) − 1;
      if edges_left(c) = 0 then S←S∪{c};
```

Figure 10

4.3 Order Correction

We locally correct invalid order numbers as we visit nodes in the graph. When grafting on a component, we are faced with two order numbers at the graft point. We choose the order number that corresponds with the subgraph of any predecessor. As we traverse the graph from b to c, we compare the order numbers of b and c. If their order numbering is not consistent with the dependency of c on b, we swap their order numbers. Should we find that b and c have the same order number, we form a slightly larger random number for the order of c by adding a random number half its bit length. While this does not result in a correct topological order, the inconsistency is limited to graft points and the resulting order gradually corrects itself after repeated evaluation. The algorithm is shown in Figure 11.

4.4 Incremental Evaluation

Given a dependency graph G and a node set S with inconsistent values, we reach a consistent assignment by applying change propagation to the elements of S, always evaluating the node in S with the lowest order number first. While this may lead to multiple evaluations of some nodes, we locally fix the ordering of the propagation area so that the order will be less likely to cause

```
procedure fix_order_numbers(b,c:graph node)
    if order(b) = order(c) then
        number_bits←⌈log2(order(b))⌉/2;
        order(c)←order(b)
            + random(number_bits) + 1
    else if order(b) > order(c) then
        temp←order(b);
        order(b)←order(c);
        order(c)←temp;
```

Figure 11

multiple evaluations in the future. This algorithm is given in Figure 12.

```
procedure evaluate(S : priority queue)
    while S ≠ ∅ do
        b←first(S);
        oldvalue←value(b);
        value(b)←semantic_equation(b);
        if value(b) ≠ oldvalue then
            for all c∈succ(b) do
                fix_order_numbers(b,c);
                S←S∪{c}
```

Figure 12

5.0 Copy bypass Propagation with Approximate Topological Ordering

The propagate algorithm in Figure 4 requires us to determine the first element of the evaluation set S in some topological order. To do this, we implement S as a priority queue and draw out the element with the lowest order number given by the approximate topological ordering scheme. There is a problem with this, however. Since we are not guaranteed that we will always get the topologically first element, we might evaluate a semantic equation that has an inconsistent CB value as an argument. This would create an invalid CB dependency.

To avoid this problem, we keep two inconsistent attribute sets. In the first set, S_1, we place copy attributes, and in the second set, S_2, we place all noncopy attributes. Since no evaluation of an attribute from S_2 can affect an evaluation from S_1, we can evaluate attributes from S_1 when it is not empty and from S_2 when S_1 is empty and S_2 is not. Thus, all CB dependencies are in place when the evaluation of noncopy attribute instances commences. The final propagation algorithm is shown in Figure 13.

```
procedure propagate(S1,S2 : priority queue)
  while (S1∪S2) ≠ ∅ do
    if S1 ≠ ∅ then
      a←first(S1);
      S1←S1−{a}
    else
      a←first(S2);
      S2←S2−{a};
    newvalue←CB_sem_func(a);
    if value(a) ≠ newvalue then
      value(a)←newvalue;
      for all b∈succ(a)∪nonlocal_succ(a) do
        fix_order_numbers(a,b);
        if is_copy_attr(b) then
          S1←S1∪{a}
        else
          S2←S2∪{a}
```

Figure 13

6.0 Reducing Storage Requirements

Two methods are given for the reduction of storage requirements. Section 6.1 gives an efficient pathkey storage algorithm that allows us to represent most pathkeys with fixed length integers. A method that allows CB values to be eliminated is discussed in Section 6.2

6.1 Efficient Pathkey Storage

An efficient method is needed for building and storing pathkeys. We can compact the string by using a binary representation for each successive index and eliminate the delimiter by appending the same number of bits for each of the successors of attribute instance a. We need $\lceil \log_2(|succ(a)|) \rceil$ bits to represent the choice. Note that we need no bits to represent a copy if succ(a) is equal to 1.

Using bit manipulation functions, we define concat to shift the previous path key $\lceil \log_2(|succ(a)|) \rceil$ bits to the left and then add the bit pattern for the next index.

$$concat_b(k,a) = shift\_left(k, \lceil \log_2(|succ(b)|) \rceil) + index(a)$$

Since we must know the length of this string of bits, we replace the null string ε with 1 in the function CB_semantic_function in Figure 3.

To test if p is a prefix of k, we shift k right by the difference in the number of bits of k and p. Say k has length lk and p has length lp. p is a prefix of k iff $lp \le lk \wedge shift\_right(k, lk - lp) = p$.

To keep the CB lookup trees in order, we will need to test pathkeys for lexigraphical order. Given two pathkeys p_1 and p_2, with bit lengths l_1 and l_2, we can test lexigraphical order as follows.

$$p_1 =_l p_2 \text{ iff } p_1 = p_2$$
$$p_1 <_l p_2 \text{ iff } l_1 \ge l_2 \wedge shift\_right(p_1, l_1 - l_2) < p_2 \vee l_1 < l_2 \wedge p_1 \le shift\_right(p_2, l_2 - l_1)$$

This allows us to reduce the size of the pathkeys, but it does not place a bound on the length of the pathkey. Each time a copy rule chain splits, the pathkey length will grow by the log of the split factor. While this split factor is small in most grammars, a bound on the pathkey length would free us from allocating variable length storage.

We can place a bound on the length of the pathkeys by introducing a few nonlocal copy dependencies into the attribute dependency graph. If creating a CB value for attribute a requires a pathkey longer than the maximum length, we install a CB dependency from the beginning of the copy chain to the predecessor p of a. We then create a CB value for a as if p were the head of a chain. The functions true_val (Figure 3) and nonlocal_succ (Figure 4) have the necessary recursion to support these nonlocal copy rules.

6.2 Elimination of Attributes Defined by Copy Rules

Provided that we can determine the predecessors of copy attribute instances, we can entirely eliminate the overhead of storing their CB values in the tree. This is done by propagating CB values in the propagate procedure and by building CB values upon demand.

When an inconsistent copy attribute is encountered in the propagate procedure (Figure 4), we do not save its value. Instead, we insert its successors into the evaluation set tagged with its CB value. This value is used when the successor is evaluated. This allows us to establish CB dependencies without storing the CB values in the tree. Notice that we do not want to insert the consistent successors of noncopy attributes into

the evaluation set as this will cause the chains which follow to be reevaluated.

When a subtree replacement is performed where there is an intersection attribute defined by a copy rule, we build the CB value. By tracing the copy chain backwards, we can locate the head of the copy chain and build the path key in reverse. This gives us exactly what we would have stored as the CB value of the intersection attribute, allowing us to remove invalid CB dependencies.

The time required to build this CB value is O(*n*Copy Chain Length*) for *n* intersection attributes. This is the same as the time required to install the new CB dependencies after the subtree modification.

The number of copy attribute instances is large. While the space occupied by the CB values is small, a great deal of storage could be saved if one avoided allocating storage for these attribute instances in addition to their values. This requires the ability to navigate through the attribute dependency graph using the underlying context free grammar and the attribute dependencies at each production.

Note that if the bounded length pathkey scheme from Section 6.1 is used, it is necessary to dynamically allocate storage for the attribute instances at the point where the pathkey length exceeds the bound.

7.0 Time and Space Bounds

First we must create the nonlocal dependencies. This is done as the attribute values initially propagate through the tree. We have used self-adjusting binary trees [ST85] to store the CB dependencies at their origin, although any concatenable queue would give comparable results. Say we have *n* nonlocal dependencies that originate from *m* attribute instances each of which have *k* nonlocal dependencies. Adding these nonlocal dependencies will cost us an amortized time of O($n\log(k)$)

If no nonlocal dependencies are altered by the subtree replacement, the incremental attribute update is done in O($|Affected_{CB}|*q*r$) where $|Affected_{CB}| = |Affected| - |Copy\ Attributes|$, q is the time required to remove the first element of the priority queue, and r is the ratio of the total number of choices made by the approximate topological ordering scheme to the number of correct choices. While the analysis of the approximate topological ordering scheme is an open problem, in our experience with attribute grammars for programming languages, the priority queue rarely exceeds several hundred elements and r is typically between 1 and 1.1. Over extended editing sessions in our implementation, the average r has been between 1.01 and 1.03.

What if nonlocal dependencies are affected by tree modification? Say we have i such dependencies originating at each of m attribute instances. Let c be the average length of a copy rule chain. If we have eliminated the copy rule attribute storage, we must spend O(c) time to rebuild each of O(m) CB values. We can perform the CB_tree_remove procedure (Figure 7) in amortized O(log(k)) time. (This is O(log(k)+i) if we have to remove each tree element.) Thus the cost of removing nonlocal dependencies after a subtree replacement is O($m(c+\log(k))$). We then need to reestablish O(m) CB dependencies, each requiring O(c) copy attribute evaluations and O(log(k)) time to insert them into the lookup tree. Since m is bound by the number of attributes at the intersection symbol, the running time of the incremental evaluation is O(($|Affected_{CB}|+c$)*q*r + log(k)).

The number of CB values that need to be stored is O(|*Copy Attributes*|). Assuming pathkeys of bounded length, this requires O(|*Copy Attributes*|) storage. Since there is at least one copy attribute instance for each nonlocal dependency, the storage used by nonlocal dependencies and pathkeys at creation attributes is also O(|*Copy Attributes*|). If the copy attribute storage is eliminated, both of these storage requirements become O(|*Copy Chains*|).

8.0 Summary

Copy bypass attribute propagation allows the dynamic replacement of copy rules with nonlocal dependencies. This is done by passing the location of the value creator and a pathkey, an encoding of the copy rule chain path, through the tree. Incremental evaluation can then be done using multiple subtree replacement. We can

store these pathkeys efficiently and place a bound on their length.

Approximate topological ordering keeps a locally correct approximation of the topological ordering of a dependency graph. This allows us to incrementally evaluate any noncircular attribute grammar with nonlocal dependencies.

The result of the combination of these strategies is an incremental evaluator for attribute grammars that has an overall performance comparable to the fastest evaluation schemes [K80, Y83], yet can evaluate all noncircular attribute grammars including those with nonlocal dependencies.

A consequence of this implementation is that a tree of the uses of aggregate values, such as symbol tables, is maintained at points where the value is modified. This tree is ordered by pathkey. We currently have an implementation that maintains two copies of this tree, one in pathkey order, and the second in order of the lookup key of the aggregate. When these aggregate values change, the difference is computed and propagation is done only to the attributes which are affected. This will be described in a future paper.

9.0 References

[DRZ85] Demers, Alan, Anne Rogers, and Frank Kenneth Zadeck. Attribute Propagation by Message Passing. *Proc. ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle, WA, June, 1985, pp. 43-59

[J84] Johnson, Gregory F. An approach to incremental semantics. TR 547, University of Wisconsin, Madison, July 1984.

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

[K80] Kastens, U. Ordered attribute grammars. *Acta Inf.* 13, 3, 1980, pp. 229-256.

[KP81] Kernighan, Brian and P. J. Plauger. *Software Tools in Pascal.* Addison-Wesley, Reading, MA,1981.

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

[RMT86] Reps, Thomas, Carla Marceau, and Tim Teitelbaum. Remote Attribute Updating for Language-based Editors. *Proc. of the 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, Jan 13-15, 1986

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

[RTD83] Reps, Thomas, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.* 5, 3, July 1983, pp. 449-477.

[ST85] Sleator, D. D. and R. E. Tarjan. Self-adjusting Binary Search Trees. *JACM 32, 3*, July 1985, pp. 652-686.

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