Sublinear-Space Evaluation Algorithms for Attribute Grammars

Thomas Reps
Alan Demers

TR 84-630
August 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work was supported in part by the National Science Foundation under grants MCS80-04218 and MCS82-02677.

Sublinear-Space Evaluation Algorithms for Attribute Grammars

Thomas Reps and Alan Demers

TR 84-630

August 1984

This work was supported in part by the National Science Foundation under grants MCS80-04218 and MCS82-02677.

Sublinear-Space Evaluation Algorithms for Attribute Grammars

THOMAS REPS and ALAN DEMERS
Cornell University

The chief hindrance to the widespread adoption of attribute-grammar-based systems has been that they are profligate consumers of storage. This paper concerns new storage management techniques that reduce the amount of storage used by reducing the number of attribute values retained at any stage of attribute evaluation; it presents one algorithm for evaluating an n-attribute tree that never retains more than $O(\sqrt{n})$ attribute values, and it presents a second algorithm that never retains more than $O(\log n)$ attribute values.

CR Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *semantics*; D.3.4 [**Programming Languages**]: Processors – *translator writing systems and compiler generators*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *denotational semantics*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: attribute grammar, attribute evaluation, language-based editors, spill files

1. INTRODUCTION

The principle drawback of attribute-grammar-based systems is their rather extravagant consumption of storage. In this paper, we present new techniques to help alleviate this problem. We present one algorithm for evaluating an n-attribute tree that never stores more than $O(\sqrt{n})$ attribute values; we present a second algorithm that never stores more than $O(\log n)$ attribute values.

Methods that have previously been suggested for making more efficient use of storage fall into several categories, and not all techniques are compatible with one another. The objective of some methods is to reduce the cost of storing the derivation tree. Schulz proposes linearizing the derivation tree so that the evaluator can make use of secondary storage for storing the tree [24]. Jazayeri and Pozefsky propose a method for constructing files of attribute-evaluation instructions during parsing as a means of completely avoiding the need to build and store the derivation tree [7]. Both of these techniques can be applied only when evaluation is carried out by an alternating-pass evaluator.

Other techniques are possible when the purpose of attribute evaluation is to obtain the value of a distinguished attribute of the derivation tree, such as one of the synthesized attributes of the root. In this situation, there are ways of reducing the amount of space devoted to storing attri-

This work was supported in part by the National Science Foundation under grants MCS80-04218 and MCS82-02677.
Authors' address: Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853.

bute values because storage does not have to be allocated for an attribute until it is defined, and storage can be reclaimed when its value has been used for the last time [16]. To reduce the amount of space needed for storing attribute values even further, Raiha proposes a method in which all attribute values in a chain of attribute instances defined by identity functions share the same storage area; after all members of one chain are no longer needed, the storage can be used for another chain [20].

These techniques are *dynamic* in the sense that decisions about when to allocate and reclaim storage depend not only on the grammar, but also on the particular tree that is being evaluated. Alternative techniques are *static* in the sense that such decisions are made at construction-time by statically analyzing the grammar. Whereas dynamic techniques can make decisions about how storage is used for individual attribute instances, static techniques must deal with classes of attributes, where usually each class consists of all of the instances of a particular attribute. In alternating-pass evaluators, for example, Jazayeri and Pozefsky propose allocating space for *all* instances of an attribute on the first pass during which *any* of the instances is defined, and reclaiming the space on the last pass during which one of the attribute instances is used [8].

This technique allocates different storage areas for different members of a class; a different approach is to use a single storage area for all members of the class. Ganzinger investigated the feasibility of implementing more than one attribute class with a single storage area. His results indicate that an automatic technique for making optimal assignments of attribute classes to storage areas would not be practical; he showed that automatically determining an allocation of attribute classes to storage areas that minimizes the number of storage areas is NP-complete [4].

This paper presents two new dynamic algorithms for evaluating a distinguished attribute of a derivation tree. These algorithms make no attempt to achieve any sort of optimal behavior, and, in contrast to most previous techniques, these algorithms may evaluate an attribute more than once; however, the characteristic that sets both algorithms apart from previous evaluation methods is that their space complexity is *sublinear, even in the worst case.* We describe one algorithm that stores at most $O(\sqrt{n})$ attribute values at any stage of evaluation, where n is the number of attributes in the derivation tree; we describe a second algorithm that stores at most $O(\log n)$ attribute values at any stage of evaluation.

The paper is organized into six sections, as follows: Section 2 introduces the terminology and notation that is used in the rest of the paper. Section 3 presents the method that allows no more than $O(\sqrt{n})$ temporary values to ever be used during evaluation. Section 4 presents the method that allows no more than $O(\log n)$ temporary values to ever be used. Section 5 discusses the relation of our work to previous work on graph pebbling. Some simple observations show that the $O(\log n)$-evaluation method is optimal in that it achieves the $\log n$ lower bound on the problem. Section 6 discusses how the methods developed in this paper can be used to increase the storage efficiency of language-based editors that use attribute grammars by allowing such editors to make use of spill files in secondary storage.

2. TERMINOLOGY AND NOTATION

An attribute grammar is a context-free grammar extended by attaching attributes to the symbols of the grammar. Associated with each production of the grammar is a set of *semantic equations*; each equation defines one attribute as the value of a *semantic function* applied to other attributes in the production. The attributes of a symbol X, denoted $A(X)$, are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. Each semantic equation defines a value for a synthesized attribute of the left-side nonterminal or an inherited attribute of a right-side symbol.

A derivation tree node labeled X defines a set of *attribute instances* corresponding to the attributes of X. A *semantic tree* is a derivation tree together with an assignment of either a value or the special token **null** to each attribute instance of the tree.

Functional dependencies among attributes in a production p or a semantic tree T can be represented by a *dependency graph*, a directed graph, denoted by $D(p)$ and $D(T)$, respectively, and defined as follows:

a) For each attribute b, the graph contains a vertex b'.

b) If attribute b is an argument of attribute c, the graph contains a directed edge (b', c').

Attribute grammars whose derivation trees have dependency graphs that contain cycles are not generally useful; thus, this paper deals only with acyclic dependency graphs.

To analyze a string according to its attribute grammar specification, first construct its semantic tree with an assignment of **null** to each attribute instance, and then evaluate as many attribute instances as possible, using the appropriate semantic equation as an assignment statement. The latter process is termed *attribute evaluation*. The order in which attributes are evaluated is arbitrary, subject to the constraint that each semantic function be evaluated only when all of its argument attributes are available.

Space utilization during attribute evaluation will be discussed under the following assumptions:

a) We assume that we have been furnished a semantic tree to evaluate, and in analyzing the amount of storage used during evaluation we will not count the storage used to represent the tree.

b) We assume that each nonterminal in the attribute grammar has at least one attribute; thus, the relationship between n, the number of attribute instances in the semantic tree, and m, the number of nodes in the semantic tree, is given by:

$$m \leq n \leq (m)(\mathrm{MaxAttrs})$$

where MaxAttrs is the maximum number of attributes of any nonterminal of the grammar.

c) We assume that a tree node contains references to each of its children, that each node is labeled with a descriptor that can be used to determine the node's arity, and that a small number of bits are associated with each node so that insertion, deletion, and membership operations on a set of tree nodes can be implemented as unit-time operations.

To simplify the presentation of the evaluation algorithms, we will assume that we have been given enough stack space to perform a recursive traversal of the semantic tree. One justification for this assumption is that the required traversals may be performed with no more that a constant amount of primary storage, using the intermediate-file scheme that is described in Section 3.4. Alternatively, the traversals may be performed (in primary storage) without using a recursion stack by using Lindstrom's pointer-rotation techniques [14] and a few additional bits at each tree node (see [21]).

In the description of the $O(\log n)$-evaluation algorithm, we will also assume that each node in the tree is labeled with its *subordinate* and *superior characteristic graphs*, denoted $r.C$ and $r.\overline{C}$, respectively. The subordinate characteristic graph at node r is the projection of the dependencies of the subtree rooted at r onto the attributes of r. To form the superior characteristic graph at node r, we imagine that the subtree rooted at r has been pruned from the semantic tree, and project the dependency graph of the remaining tree onto the attributes of r. The vertices of the characteristic graphs at r correspond to the attributes of r; the edges of the characteristic graphs at r correspond to transitive dependencies among r's attributes.

To define these graphs more precisely, we make the following definitions:

a) Given directed graphs $A = (V_A, E_A)$ and $B = (V_B, E_B)$, that may or may not be disjoint, the *union* of A and B is defined as:

$$A \cup B = (V_A \cup V_B, E_A \cup E_B)$$

b) The *deletion* of B from A is defined as:

$$A - B = (V_A, E_A - E_B)$$

Note that deletion deletes only edges.

c) Given a directed graph $A = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of A onto V' is defined as:

$$A / V' = (V', E')$$

where $E' = \{(v, w) \mid v, w \in V'$ and there exists a path from v to w in A that does not contain any elements of $V'\}$.

Formally, let r be a node in semantic tree T, and let the subtree rooted at r be denoted T_r; the subordinate and superior characteristic graphs at r are defined by:

$$r.C \equiv D(T_r)/A(r)$$

$$r.\overline{C} \equiv (D(T) - D(T_r))/A(r)$$

The subordinate characteristic graphs can be constructed during a single endorder traversal of the tree; the superior characteristic graphs can then be constructed during a single preorder traversal of the tree. Generating the characteristic graphs requires a linear amount of processing time; if represented as dependency matrices, each matrix requires no more than MaxAttrs^2 bits at each tree node.

3. AN EVALUATION ALGORITHM THAT STORES AT MOST O($\sqrt{n}$) ATTRIBUTES

This section describes a space-efficient method for evaluating a distinguished attribute of an n-attribute tree. At any one time, this method stores at most $O(\sqrt{n})$ attribute values, and uses at most $O(\sqrt{n})$ units of additional space; altogether, the method performs at most $O(n)$ function applications. For each of these terms, the constant of proportionality depends only on the quantities MaxAttrs and MaxSons (the maximum number of nonterminals on the right-side of any production in the attribute grammar), both of which are constants for a given grammar.

The $O(\sqrt{n})$-evaluation method is a divide-and-conquer algorithm that makes use of the concept of a separator set: a set of vertices A is a *separator set* of a graph G if we can partition the vertices of G into A and two other sets, B and C, such that no edge connects a vertex of B to a vertex of C, or vice versa. We say that A is a *k-separator set* if we can partition the vertices of G into A and k other sets $B_1 \cdots B_k$, such that no edge connects a vertex of B_i to a vertex of B_j, for $i \neq j$.

The $O(\sqrt{n})$-evaluation algorithm relies on the fact that attribute dependency graphs contain small separator sets. Because each attribute instance in a semantic tree depends only on attribute instances in a single production instance, the only attributes in the subtree rooted at node X that can depend on attributes outside the subtree at X belong to $A(X)$; similarly, the only attributes outside the subtree rooted at X that can depend on attributes in the subtree rooted at X belong to $A(X)$. Thus $A(X)$, the set of attribute instances associated with node X, is a separator set of the tree's dependency graph. The attributes of *each* tree node is a separator set of the dependency graph, and each of these separator sets is bounded in size by MaxAttrs.

The existence of these separator sets allows attribute evaluation to be carried out using a divide-and-conquer strategy. The task of evaluating a distinguished attribute b is divided into smaller tasks by first finding a separator set of the dependency graph, and then letting each separator-set element be the distinguished attribute of a component of the dependency graph. On the completion of each of these subproblems, the value of the distinguished attribute is saved, but the storage for all other attribute values in the component is reclaimed.

To ensure that the work involved in solving each subproblem is confined to a single component of the dependency graph, the subproblems are treated in an order that respects the order of separator set elements in the partial order given by $D(T)$. When it comes time to solve the subproblem with distinguished attribute c of the separator set, every ancestor of c that belongs to the separator set has already been evaluated. It will be shown that the arguments of c either belong to the separator set, or else they are contained in exactly one component of $D(T)$, cut off from the rest of $D(T)$ by separator-set elements; consequently, each subproblem is no larger than a single component of $D(T)$.

The partitioning step of the $O(\sqrt{n})$-evaluation method produces a group of $O(\sqrt{n})$ subproblems, each of size $O(\sqrt{n})$. A problem of size $O(\sqrt{n})$ can be solved using at most $O(\sqrt{n})$ attribute values by evaluating attributes in a topological order of the dependency graph, without discarding any values. Once a subproblem is solved, we can release the space used to store all values of the subproblem, except for the value of the distinguished attribute. Because there are at most

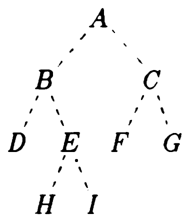
$O(\sqrt{n})$ subproblems, there are at most $O(\sqrt{n})$ separator-set values to be saved; thus, at most $O(\sqrt{n})$ values are needed altogether.

Note that the $O(\sqrt{n})$-evaluation algorithm is not recursive; each of the subproblems is solved in a straightforward manner without partitioning the problem still further. Recursion is used in the algorithm described in Section 4, which stores at most $O(\log n)$ attribute values at any one time; however, the $O(\sqrt{n})$-evaluation algorithm takes no more than a linear number of steps, whereas the recursive, $O(\log n)$-evaluation algorithm uses a polynomial number of steps.

The $O(\sqrt{n})$-evaluation algorithm has three parts: a *partitioning step*, a *projection step*, and an *evaluation step*. The partitioning step, described in Section 3.1, finds a suitable collection of subproblems to solve. The projection step, described in Section 3.2, generates an order in which to solve the subproblems that ensures that each subproblem is smaller than the original problem. The evaluation step, described in Section 3.3, then simply solves the subproblems in the required order.

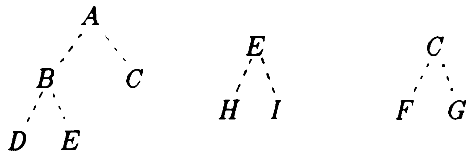
3.1. Partitioning

Rather than partitioning the problem according to the structure of the dependency graph, we partition the problem according to the structure of the derivation tree. A set of nodes P partitions a tree T into *components*, where a component is a maximal-size, connected region of T in which none of the interior nodes are elements of P. For example, the set E, C partitions the tree:



into the three components:

(1) (2) (3)



The goal of the partitioning step is to divide the derivation tree into $O(\sqrt{n})$ components that are each no larger than $O(\sqrt{n})$ nodes and that each adjoin at most MaxSons + 1 other components. This is accomplished in two phases: the first phase partitions the derivation tree into components of maximum size $O(\sqrt{m})$, where m is the number of nodes in the tree; the second phase further subdivides the components found during the first phase so that each component adjoins at most MaxSons + 1 other components.[1]

Although each phase subdivides the tree according to m, each node can have no more than MaxAttrs attributes, so m is related to n by:

$m \leq n \leq (m)(\text{MaxAttrs})$

Consequently,

$$\sqrt{m} \leq \sqrt{n} \leq (\sqrt{m})(\sqrt{\text{MaxAttrs}})$$

and $O(\sqrt{m}) = O(\sqrt{n})$.

Phase 1. The first phase of partitioning is carried out by the procedure Partition, stated as Algorithm 1. Partition performs a scan of the tree, starting from the root, partitioning the tree into components, each no larger that $\sqrt{m} + 1$ nodes. Partition constructs the set PartitionSet bottom-up, by traversing the nodes of T in endorder with the recursive procedure PartitionBySize. PartitionSet is initially empty, and PartitionBySize adds a new node to PartitionSet whenever a node is found that is at the root of a component larger than $\sqrt{m}$/MaxSons.

For each node Y, PartitionBySize computes how large a component there would be if Y were the root of a component. Each recursive call on PartitionBySize(Y_i), where Y_i is the i^{th} child of Y, either returns 1, if Y_i is a leaf or if Y_i belongs to PartitionSet, or it returns how large a com-

```
Partition(T):
  let
    T = a derivation tree
    m = an integer
    PartitionSet = a set of tree nodes
  in
    PartitionSet := φ
    m := the number of nodes in T
    PartitionBySize(root(T))
    return(PartitionSet)

PartitionBySize(Y)
  let
    Y = a tree node
    Y_i = the i^th child of Y
    L, i = integers
    PartitionSet = a set of tree nodes (global)
    m = an integer (global)
  in
    L := 1
    for i := 1 to NumberOfChildren(Y) do
      L := L + PartitionBySize(Y_i)
      od
    if L > √m /MaxSons then
      Insert Y into PartitionSet
      L := 1
    return(L)
```

Algorithm 1. Partition T into components of size $O(\sqrt{m})$, where m is the number of nodes in T.

[1]In practice, one would probably combine the two phases and carry out the partitioning of the tree in a single step.

ponent there would be if Y_i were the root of a component; thus, the quantity returned from each call ranges from 1 to $\sqrt{m}$ /MaxSons. The size of the component rooted at Y is computed by adding 1 (for Y itself) to the sum of the values returned by the calls to PartitionBySize with Y's children. Because Y has at most MaxSons children, the number of nodes in the component can be no larger than $\sqrt{m} + 1$. Because each node is visited once during the endorder traversal of T, the total running time of Algorithm 1 is $O(m)$.

Except for the component containing the root of the tree, the root of each component is shared by exactly one other component; thus, for the set of components C_P, defined by a partition P, the sum of the components' sizes is related to the total number of nodes m by:

$$m = 1 + \sum_{c \epsilon C_P} (|c| - 1)$$

Except possibly for the component containing the root of the tree, each component defined by the set PartitionSet of Algorithm 1 contains at least $\sqrt{m}$ /MaxSons + 1 nodes, so

$$m \geq 1 + (|C_{\text{PartitionSet}}| - 1)(\sqrt{m} / \text{MaxSons})$$

Solving for $|C_{\text{PartitionSet}}|$, we get:

$$|C_{\text{PartitionSet}}| \leq \frac{(m-1)\text{MaxSons}}{\sqrt{m}} + 1$$

$$< (\text{MaxSons})(\sqrt{m}) + 1$$

Thus, the first phase of partitioning finds no more than (MaxSons)($\sqrt{m}$) + 1 components, each containing no more than $\sqrt{m} + 1$ nodes.

The only operations performed on PartitionSet are insertions and, in the next phase, membership tests, so PartitionSet can be implemented using a single bit at each tree node and no other additional storage.

Phase 2. The components found during the first phase of partitioning may be adjacent to as many as $O(\sqrt{m})$ other components. An additional partitioning phase is needed because this condition would cause the projection step described in the next section to use too much storage. The second phase of partitioning further subdivides the components from the first phase of partitioning by adding to the partition set every node that is a lowest common ancestor of some pair of partition nodes. This process makes each component adjacent to no more than MaxSons + 1 other components, and at most doubles the size of the old partition set.

The procedure AddAncestors, stated as Algorithm 2, constructs the set NewPartition bottom-up by considering the components defined by OldPartition in endorder, traversing each component with the recursive routine AddAncestorsInComponent. NewPartition is initially empty, and AddAncestorsInComponent adds a new node to NewPartition whenever a node is found that is the lowest common ancestor of some pair of nodes that belong to OldPartition. Note that because for each node X, LowestCommonAncestor(X, X) = X, every node in OldPartition is put in NewPartition.

For each node Y, AddAncestorsInComponent checks if Y is the lowest common ancestor of some pair of nodes that belong to OldPartition. Each recursive call on

```
AddAncestors(T, OldPartition):
  let
    T = a derivation tree
    OldPartition, NewPartition = sets of tree nodes
    X = a tree node
  in
    NewPartition := ϕ
    for each node X in an endorder traversal of T do
      if X ϵ OldPartition then
        AddAncestorsInComponent(X)
        Insert X into NewPartition
      od
    AddAncestorsInComponent(root(T))
    return(NewPartition)

AddAncestorsInComponent(Y)
  let
    Y = a tree node
    Y_i = the i^th child of Y
    i, SubordinatePartitionNodes = integers
  in
    if Y ϵ NewPartition then return(true)
    SubordinatePartitionNodes := 0
    for i := 1 to NumberOfChildren(Y) do
      if AddAncestorsInComponent(Y_i) = true then
        SubordinatePartitionNodes := SubordinatePartitionNodes + 1
      od
    if SubordinatePartitionNodes = 0 then return(false)
    else if SubordinatePartitionNodes = 1 then return(true)
    else
      Insert Y into NewPartition
      return(true)
```

Algorithm 2. Given OldPartition, a set of nodes in T, build the set NewPartition, where NewPartition $= \{Z \mid Z =$LowestCommonAncestor(X, Y), where $X, Y \epsilon$OldPartition$\}$

AddAncestorsInComponent(X_i), returns **true** if the subtree rooted at X_i contains a member of OldPartition. If this is true of more than one child of X, then X is the lowest common ancestor of some pair of nodes in OldPartition, and is added to NewPartition.

The partition set computed by this process has two important properties:

a) The number of elements in NewPartition is at most double the number of elements in OldPartition.

b) Each component defined by NewPartition is adjacent to at most MaxSons + 1 other components.

Because each member of OldPartition is made a member of NewPartition, each of the calls on AddAncestorsInComponent made by AddAncestors is confined to a single one of the components defined by OldPartition. Each member of OldPartition causes one call on AddAncestorsInComponent; thus, the total running time of Algorithm 2 is $O(m)$.

3.2. Projection

The $O(\sqrt{n})$-evaluation algorithm uses a divide-and-conquer strategy to evaluate a distinguished attribute of a derivation tree. This problem is divided into subproblems using the methods described in the previous section to partition the tree. Each subproblem involves evaluating an attribute of one of the partition set nodes, treating it as the distinguished attribute of a component of the tree.

To ensure that solving each subproblem involves only evaluations of attributes that are part of a single component, the order in which subproblems are solved must respect the order of the distinguished attributes of the subproblems in the partial order given by the dependency graph $D(T)$. By treating the subproblems in this order, when it comes time to solve the subproblem with distinguished attribute b, every ancestor of b that is an attribute of the partition set P already has a value. Because an attribute is defined solely in terms of attributes of a single production, the arguments of b are either members of $A(P)$, or they are contained in exactly one component of $D(T)$, cut off from the rest of $D(T)$ by members of $A(P)$; consequently, the work involved in solving each subproblem is confined to a single component.

This order is found by constructing the graph $D(T)/A(P)$ with the procedure Project, which is stated below as Algorithm 3. Project constructs the graph $D(T)/A(P)$ by taking each of the components defined by P, projecting the dependency graph of the component onto the attributes of the partition nodes in the component, and merging the projected graphs together. When the number of nodes in P is no more than $O(\sqrt{m})$, the size of the graph $D(T)/A(P)$ is no more than $O(\sqrt{m})$; thus, the total space cost of Algorithm 3 is $O(\sqrt{m})$.

```
Project(T, P):
  let
    T = a derivation tree
    P, Q = sets of tree nodes
    G, H = directed graphs
    X = a tree node
    K_X = the component rooted at X
  in
    G := (φ, φ)
    for each node X in an endorder traversal of T do
      if X ε P or X = root(T) then
         Q := the nodes of K_X that are in P
         H := D(K_X)/A(Q)
         G := G ∪ H
      od
    return(G)
```

Algorithm 3. Build the graph $G := D(T)/A(P)$.

The projection operation for each component can be done by a depth-first search from each inherited attribute of the root of the component and each synthesized attribute of partition nodes at leaves of the component. When each component is no larger than $O(\sqrt{m})$, and when there are no more than MaxSons + 1 partition nodes that are part of each component, the running time of this operation is no more than $O(\sqrt{m})$; given $O(\sqrt{m})$ components, the total running time of Algorithm 3 is $O(m)$.

3.3. The Attribute Evaluation Algorithm

The procedures described in the previous two sections are used by the procedure Evaluate, stated as Algorithm 4, to divide an evaluation problem of size n into a group of $O(\sqrt{n})$ subproblems of maximum size $O(\sqrt{n})$. Algorithms 1 and 2 are used to find a set of nodes P whose attributes $A(P)$ are an $O(\sqrt{n})$-separator set of the dependency graph $D(T)$. Algorithm 3 is used to construct the graph $G := D(T)/A(P)$, and by topologically sorting G, we find an order for the members of $A(P)$ that respects their order in $D(T)$. This order is used to schedule calls on DemandValue, a recursive procedure for evaluating an attribute by making recursive calls on DemandValue for all of the attribute's unevaluated predecessors.

```
Evaluate(T, b):
 let
   T = a derivation tree
   b, c = attribute instances in T
   P = a set of tree nodes
   G = a directed graph
   S = an ordered list of attribute instances
 in
   P := AddAncestors(T, Partition(T))
   G := Project(T, P)
   S := TopologicalSort(G)
   for each attribute c in the order listed in S do
     DemandValue(c)
     Release storage for all attributes that are not members of S
     od
   DemandValue(b)

DemandValue(b)
  let
    b, c = attribute instances
  in
    while there exists c, an unevaluated argument of b do
      DemandValue(c)
      od
    evaluate b
```

Algorithm 4. Given a semantic tree T and a distinguished attribute b, evaluate b.

Because the evaluation order of the separator-set elements respects their order in $D(T)$, when attribute c is evaluated, every ancestor of c that is a separator-set element has been given a value. This restricts each semantic function application made during a call on DemandValue to a single component of $D(T)$, so at most $O(\sqrt{n})$ steps are used to evaluate at most $O(\sqrt{n})$ attributes. Only the values of separator-set elements are retained after each call on DemandValue; because Evaluate makes $O(\sqrt{n})$ calls on DemandValue, the total number of values retained at any time during evaluation is bounded by $O(\sqrt{n})$, and the total number of steps used is bounded by $O(n)$.

3.4. Discussion

In connection with implementing spill files (see Section 6) in the Synthesizer Generator [22], the first author has implemented two versions of the $O(\sqrt{n})$-evaluation algorithm. One version is essentially what is described above. The second version is an adaptation of the method for grammars in the class of ordered attribute grammars [9]. In the latter version, an explicit projection step is avoided by making use of the total order on the tree's attributes that is implicit in the finite-state-machine descriptions (plans) that make up an ordered-attribute-grammar evaluator. The topological order on the separator-set elements used in Algorithm 4 is generated during an interpretation of the plans that skips over non-separator-set elements.

It is interesting to compare the $O(\sqrt{n})$-evaluation method with the method proposed by Schulz for using secondary storage in an alternating-pass evaluator [24]. Schulz pointed out that by using a linearized representation of derivation trees, an alternating-pass evaluator needs only three intermediate files in order to evaluate a tree's attributes. On every odd-numbered pass, the tree is stored in left-to-right preorder, and on every even-numbered pass, it is stored in right-to-left preorder. The evaluator performs alternating left-to-right and right-to-left passes over the tree by reading the preorder representation from an intermediate file, stacking the "stalk" of the tree in a second file, and writing out a postorder representation to a third file. However, left-to-right postorder is equivalent to right-to-left preorder, and right-to-left postorder is equivalent to left-to-right preorder, which means that at the end of each pass the output file is ready for the next pass in the reverse direction. Thus, only a constant amount of primary storage is ever needed for Schulz's method.

The $O(\sqrt{n})$-evaluation method provides a rather different way of making efficient use of primary and secondary storage. If each of the components defined by the partition P in Algorithm 4 is linearized and stored in secondary storage, then the remainder of the algorithm need never have more than one component in primary storage at any time, so no more than $O(\sqrt{n})$ units of primary storage are needed to carry out evaluation. Because P is generated with an endorder tree traversal, the partitioning step may be implemented with Schulz's intermediate-file scheme, and thus requires no more than a constant amount of primary storage to generate. (These observations justify the assumptions made in Section 2 to count neither the storage used to represent the tree nor the stack space needed for traversing the tree).

It is not apparent that either of the two methods is always faster than the other. To use Schulz's technique, the alternating-pass evaluator is unable to skip over subtrees in which no attributes will be evaluated; the entire tree must be traversed on each pass. As Farrow complains, it can be "... irksome to discover that one pass of the evaluator may be spent doing little but turning the tree around" [3]. In contrast, the $O(\sqrt{n})$-evaluation method is guaranteed to make progress during each subproblem. On the other hand, the $O(\sqrt{n})$-evaluation method may be inefficient because values of attributes calculated in order to evaluate one distinguished attribute may be thrown away only to be recalculated when it comes time to evaluate some other distinguished attribute. This sort of inefficiency can probably be reduced by clustering subproblems so that several distinguished attributes, rather than just a single distinguished attribute, can be evaluated when a given component is resident in primary storage.

Schulz's method has the additional drawback that there exist some grammars that cannot be evaluated in any fixed number of passes [11], and so the method is applicable only to a subclass of the noncircular attribute grammars. In contrast, the $O(\sqrt{n})$-method works for any noncircular grammar.

4. AN EVALUATION ALGORITHM THAT STORES AT MOST O(log n) ATTRIBUTES

This section describes an algorithm for evaluating a distinguished attribute of an n-attribute tree that stores at most $O(\log n)$ attribute values at any stage of evaluation. The algorithm uses a recursive divide-and-conquer strategy; as with the $O(\sqrt{n})$-evaluation algorithm, the problem is divided into a partitioning step, a projection step, and an evaluation step, described in Sections 4.1, 4.2, and 4.3, respectively.

We will assume that each node in the tree is labeled with its subordinate and superior characteristic graph. The subordinate characteristic graphs can be constructed during a single endorder traversal of the tree; the superior characteristic graphs can then be constructed during a single preorder traversal of the tree. As we stated in Section 2, generating the characteristic graphs requires a linear amount of processing time, and if represented as dependency matrices, each matrix requires no more than MaxAttrs^2 bits at each tree node.

4.1. Partitioning

Because the $O(\log n)$-evaluation algorithm is recursive, the partitioning step is applied to a component of the tree, rather than to the entire tree. The goal of the partitioning step is to find a set of nodes that partitions a component into at most 2MaxSons^2 subcomponents, each of which contains no more than $m/2\text{MaxSons} + 1$ nodes, where m is the number of nodes in the original component.

Partitioning is carried out by the procedure Partition, stated below as Algorithm 5. Algorithm 5 is very much like Algorithm 1, the first phase of partitioning used in the $O(\sqrt{n})$-evaluation method, except that:

a) Algorithm 5 partitions a component K, defined by the set P, rather than the entire tree,[2] and

[2] P represents the context in which a call on Partition is made. K is a connected region of the tree defined by nodes in P; the root of K is in P, and some of the leaves of K are also in P. For now, it is easier to ignore P altogether, and think of K itself as a tree.

```
Partition(K, P):
  let
    K = a component of the derivation tree
    P, Q = sets of tree nodes
    X = a tree node
    m = an integer
  in
    Q := φ
    m := the number of nodes in K
    PartitionBySize(root(K))
    return(Q)

PartitionBySize(Y)
  let
    Y = a tree node
    Y_i = the i^th child of Y
    L, i = integers
    P, Q = sets of tree nodes (global)
    m = an integer (global)
  in
    L := 1
    if Y ∉ P then
      for i := 1 to NumberOfChildren(Y) do
        L := L + PartitionBySize(Y_i)
        od
      if L > m/2MaxSons^2 then
        Insert Y into Q
        L := 1
    return(L)
```

Algorithm 5. Given a component K that has m nodes, where K is defined by the set P, find a set of nodes that partitions K into no more than 2MaxSons^2 components, each containing no more than $m/$2MaxSons + 1 nodes.

b) Algorithm 5 partitions K into a constant number of sub-components.

Partition performs a scan of the K, starting from its root, partitioning K into sub-components no larger than $m/$2MaxSons + 1 nodes. Partition constructs a partition set Q bottom-up, by traversing the nodes of K in endorder with the recursive procedure PartitionBySize. Q is initially empty, and Partition adds a new node to Q whenever a node is found that is at the root of a sub-component larger than $m/$2MaxSons^2 nodes.

For each node Y, PartitionBySize computes how large a component there would be if Y were the root of a sub-component; PartitionBySize performs an endorder traversal of the subtree rooted at Y, descending no further when encountering a node that is a member of P. Each child of Y contributes from 1 to $m/$2MaxSons^2 nodes to this total; thus, the size of the component rooted at Y is no larger than $m/$2MaxSons + 1. The total running time of Algorithm 5 is $O(m)$.

We can show that the number of components found by Algorithm 5 is no more than 2MaxSons^2 by the following argument: The sum of the sizes of the components K_Q, defined by the partition Q found by Algorithm 5, is related to the total number of nodes m by:

$$m = 1 + \sum_{k \in K_Q} (|k|-1)$$

Each component, except possibly the one containing the root of the tree, contains at least $m/2\text{MaxSons}^2 + 1$ nodes, so

$$m \geq 1 + (|K_Q|-1)(m/2\text{MaxSons}^2)$$

Solving for $|K_Q|$, we get:

$$|K_Q| \leq \frac{(m-1)(2\text{MaxSons}^2)}{m} + 1$$

$$< 2\text{MaxSons}^2 + 1$$

Thus, the partitioning step finds no more than 2MaxSons^2 components, each containing no more than $m/2\text{MaxSons} + 1$ nodes.

4.2. Projection

The $O(\log n)$-evaluation algorithm treats each of the attributes of the partition-set nodes found by the partitioning step as the distinguished attribute of a subproblem. As in the $O(\sqrt{n})$-evaluation algorithm, it is necessary to ensure that the work required to solve each of the subproblems involves only evaluations of attributes that are part of a single component. To do this, the subproblems must be solved in an order that respects the order of the distinguished attributes of the subproblems in the partial order given by the dependency graph $D(T)$.

If Q is the partition set that defines these subproblems, the proper order is found by constructing the graph $G = D(T)/A(Q)$ with the procedure Project, stated as Algorithm 6. As with the partitioning step, the $O(\log n)$-evaluation method differs from the corresponding step of the $O(\sqrt{n})$-evaluation method because it deals with components rather than the entire tree. One difference between Algorithm 6 and Algorithm 3 is that Algorithm 6 uses the characteristic graphs to represent the context in which a projection takes place.

As is the case with the partitioning algorithm given in the previous section, the set P represents a context in which Project is called, and defines the component K. Q is a set of nodes that partitions K, i.e. all elements of Q are nodes of K. Project constructs $D(T)/A(Q)$ bottom-up, by considering each node of K in endorder. Let T_X denote the subtree of T rooted at X. For each node X in K, Project computes a graph G_X that is the projection of $D(T_X)$ onto R, where R is the set $X \cup \{\text{all nodes of } Q \text{ that are in } T_X\}$. Given the graphs G_{X_i} for all sons X_i of X, the computation of G_X takes no more than a constant amount of time and space.

Because Q is no larger than 2MaxSons^2, G_X contains no more than $4(\text{MaxAttrs}^2)(\text{MaxSons}^4)$ vertices. Consequently, G_X can be represented using a dependency matrix that contains no more than $4(\text{MaxAttrs}^2)(\text{MaxSons}^4)$ bits.

```
Project(K, P, Q):
  let
    K = a component of the derivation tree
    P, Q, R = sets of tree nodes
    G_X = a directed graph associated with node X
    X = a tree node
    X_i = the i^th child of X
  in
    for each node X in an endorder traversal of K do
      if X ∈ P and X ≠ root(K) then G_X := X.C
      else
        p := the production that applies at node X
        R := X ∪ the nodes of the component rooted at X that are in Q
        G_X := [[ ⋃_{x ∈ X_i} G_x ] ∪ D(p)] / A(R)
      od
    return(G_root(K) ∪ root(K).C̄)
```

Algorithm 6. Given a component K, where K is defined by the set P, and a partition of K named Q, build the graph $G = D(T)/A(Q)$.

Project makes a single pass over K, and does a constant amount of work at each node, so the total running time is $O(|K|)$; its storage requirement is bounded by a constant, plus $4(\text{MaxAttrs}^2)(\text{MaxSons}^4)$ bits at each node of the tree.

4.3. The Attribute Evaluation Algorithm

The attribute evaluation algorithm stated as Algorithm 7 consists of the mutually recursive procedures DivideAndConquer and DemandAncestors. DivideAndConquer solves the problem of evaluating a distinguished attribute b in a component K defined by the partition set P. As the name suggests, a divide-and-conquer strategy is used to break the problem up into smaller problems. The method used is first to call the procedure Partition, given as Algorithm 7, to find a set of nodes Q that partitions K into smaller components, and then let the attributes of $A(Q)$ be distinguished attributes of smaller problems. By the argument given in Section 4.1, each of these components is no larger than $|K|/2\text{MaxSons} + 1$.

The argument P in DivideAndConquer is a set of nodes consisting of all nodes found by Partition in active calls to DivideAndConquer. To confine the amount of work to a single one of the components defined by P, whenever DivideAndConquer is called to evaluate attribute b, all attributes belonging to $A(P)$ that are ancestors of b have values. Initially P is the empty set, so the condition is trivially established by Evaluate.

To establish this condition for the subproblems that result from the call to Partition, DemandAncestors orders the calls to solve subproblems so that they respect the order of attributes in $D(T)$; this is accomplished by passing DemandAncestors the graph $G = D(T)/A(Q)$, found by Project. DemandAncestors causes values to be given to unevaluated ancestors of b in G by a recursive demand-evaluation process initiated on the unevaluated predecessors of b in G.

```
Evaluate(T, b):
  let
    T = a derivation tree
    b = an attribute instance
  in
    DivideAndConquer(b, φ)

DivideAndConquer(b, P)
  let
    b = an attribute instance
    P, Q = sets of tree nodes
    G = a directed graph
    T = a derivation tree (global)
    K = a component of T
  in
    K := the component of T defined by P containing b's arguments
    if |K| ≤ 2MaxSons² then DemandValue(b)
    else
      Q := Partition(K, P) ∪ TreeNode(b)
      G := Project(K, P, Q)
      DemandAncestors(b, G, P ∪ Q)

DemandAncestors(b, G, P):
  let
    b, c = attribute instances
    G = a directed graph
    P = a set of tree nodes
  in
    while there exists c, an unevaluated predecessor of b in G do
      DemandAncestors(c, G, P)
      od
    DivideAndConquer(b, P)
    Release storage for attributes that do not belong to A(P)
```

Algorithm 7. Given a semantic tree T and a distinguished attribute b, evaluate b.

Because G is no larger than 2(Maxattrs)(MaxSons^2), the number of recursive calls made is bounded by 4(Maxattrs)(MaxSons^2).

Because each call on Partition divides a component into no more than 2MaxSons^2 subcomponents, the number of subproblems is no more than 2(Maxattrs)(MaxSons^2). Because the problem is subdivided until components are no larger than 2MaxSons^2, no more than 2(Maxattrs)(MaxSons^2) attributes are needed to evaluate the distinguished attribute of a component by the call to DemandValue. Thus, the number of values needed to evaluate a tree of size m is described by the recurrence equation:

$$V(m) \leq \begin{cases} 2(\text{MaxAttrs})(\text{MaxSons}^2) & \text{for } m \leq 2\text{MaxSons}^2 \\ V(\frac{m}{2\text{MaxSons}} + 1) + 2(\text{MaxAttrs})(\text{MaxSons}^2) & \text{for } m > 2\text{MaxSons}^2 \end{cases}$$

which has the solution $V(m) = O(\log m)$.

The number of steps used by Algorithm 7 is described by:

$$T(m) \leq \begin{cases} 2(\text{MaxAttrs})(\text{MaxSons}^2) & \text{for } m \leq 2\text{MaxSons}^2 \\ (\text{MaxAttrs}) \sum\limits_{k \epsilon K_Q} T(|k|) + bm & \text{for } m > 2\text{MaxSons}^2 \end{cases} \tag{1}$$

where K_Q is the set of components defined by Q that partitions K in Algorithm 7, so that

$$1 \leq |k| \leq \frac{m}{2\text{MaxSons}} + 1, \quad \sum_{k \epsilon K_Q} (|k|-1) \leq m, \quad \text{and} \quad |K_Q| \leq 2\text{MaxSons}^2$$

An inductive argument is used to show that this equation has the polynomial solution $T(m) = O(m^{2+\log_{\text{MaxSons}} 2\text{MaxAttrs}})$.

Proof. Let the components of K_Q be numbered from 1 to $|K_Q|$ so that they may be indexed by an integer i, $1 \leq i \leq |K_Q|$. We use m_i to denote the size of the i^{th} component. For brevity, let A denote MaxAttrs and S denote MaxSons. With this notation, we can rewrite equation (1) as:

$$T(m) \leq \begin{cases} 2AS^2 & \text{for } m \leq 2S^2 \\ A\sum\limits_i T(m_i) + bm & \text{for } m > 2S^2 \end{cases} \tag{2}$$

where

$$1 \leq m_i \leq \frac{m}{2S} + 1, \quad \sum_i (m_i - 1) \leq m, \quad \text{and} \quad i \leq 2S^2$$

Since there are at most $2S^2$ components, each no larger than $m/2S + 1$, and since $m/2S + 1 \leq m/S$, for $m \geq 2S$, we can replace equation (2) by

$$T(m) \leq \begin{cases} 2AS^2 & \text{for } m \leq 2S^2 \\ 2AS^2 T(\frac{m}{S}) + bm & \text{for } m > 2S^2 \end{cases}$$

Let c be large enough so that, for all $\hat{m} \leq 2S^2$,

$$T(\hat{m}) \leq c\hat{m}^{2+\log_S 2A} + \frac{b}{1-2AS}\hat{m}$$

Inductive assumption: Assume that, for all $\hat{m} < m$,

$$T(\hat{m}) \leq c\hat{m}^{2+\log_S 2A} + \frac{b}{1-2AS}\hat{m}$$

Then,

$$T(m) \leq 2cAS^2\left(\frac{m}{S}\right)^{2+\log_S 2A} + \frac{2bAS^2}{1-2AS}\left(\frac{m}{S}\right) + bm$$

$$\leq 2cAm^{2+\log_S 2A}\left(\frac{1}{S}\right)^{\log_S 2A} + \frac{2bASm}{1-2AS} + bm$$

$$\leq 2cAm^{2+\log_S 2A}\left(\frac{1}{2A}\right) + \frac{2bASm + bm - 2bASm}{1-2AS}$$

$$\leq cm^{2+\log_S 2A} + \frac{b}{1-2AS}m$$

which establishes the inductive assumption for m. Thus, the recurrence equation has the solution $T(m) = O(m^{2+\log_S 2A})$.

5. RELATION TO GRAPH PEBBLING

As originally employed by Knuth, an attribute grammar specifies the translation of a context-free language into a semantic domain. The attribute grammar implicitly defines how each attribute is assigned a value; the meaning of a string is defined to be the value of a distinguished synthesized attribute of the root of the string's semantic tree. In this situation, the evaluator must produce the value of the meaning attribute, but need not retain the value of any of the other attributes. This section discusses the relevance of certain graph pebbling results to the problem of evaluating a distinguished attribute.

The *(repebbling-allowed) pebble game* has been used to analyze the resource requirements of a wide range of computational situations.[3] The game has two rules:

a) A vertex can be pebbled whenever there are pebbles on all of its predecessors.

b) A pebble can be removed at any time.

and the objective is to pebble a distinguished vertex of the graph while obeying the constraint that at any time at most a given number of vertices are pebbled.

Attribute evaluation can be modeled as a pebbling game on a tree's dependency graph, where each pebble corresponds to a stored attribute value, and where a step of the game that pebbles a vertex corresponds to a semantic function evaluation. Thus the maximum number of vertices pebbled simultaneously during the game tells us the number of temporary values required, and the number of steps taken during the game tells us the number of semantic function evaluations required.

The reader should understand that these measures do not correspond exactly to the space and time requirements of attribute evaluation; pebbling costs give a precise measure of the space and time costs only when the following three conditions are met:

---

[3] [19] is an excellent survey of the literature on pebbling.

a) Each semantic function application requires a constant amount of time and space.

b) The space required to store each attribute value is bounded by a constant.

c) Pebbles account only for the space used to store attribute values and fail to account for all other space used, such as that used for deciding the order in which attributes are to be evaluated. The amount of additional space used must be of the same order as the number of pebbles used.

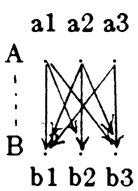
Despite the fact that these conditions are not necessarily true in practice, the pebbling game is still a useful heuristic method for studying attribute evaluation; even when the space required to store each attribute value is not bounded by a constant, an $o(n)$ pebbling method conserves storage by reducing the number of values that need to be retained during evaluation.

Many attribute evaluation algorithms pebble the vertices of a dependency graph in topological order without removing any pebbles [12, 13, 10]; an n-vertex dependency graph is pebbled in n steps using n pebbles. This paper discusses pebbling a vertex of a dependency graph using $o(n)$ pebbles by possibly using more than n steps.[4] The algorithm presented in Section 3 uses $O(\sqrt{n})$ pebbles and no more than $O(n)$ steps. The algorithm presented in Section 4 uses only $O(\log n)$ pebbles but, as will be shown below, requires possibly a non-linear number of steps.

It has previously been established that an arbitrary directed acyclic graph can be pebbled with $O(n/\log n)$ pebbles [6], and that there exist graphs that require $\Omega(n/\log n)$ pebbles[5] [18]. It has also been established that planar graphs can be pebbled with $O(\sqrt{n})$ pebbles [15], and that there exist graphs that require $\Omega(\sqrt{n})$ pebbles [1]. These lower-bound results, however, are not applicable to attribute evaluation because attribute grammars allow only limited kinds of dependencies between attributes; neither the graphs that establish the $\Omega(\sqrt{n})$ lower bound for planar graphs, nor the graphs that establish the $\Omega(n/\log n)$ lower bound for arbitrary graphs are possible attribute dependency graphs.

Attribute dependency graphs are not, in general, planar; consequently, they cannot, in general, be pebbled by the $O(\sqrt{n})$ method for pebbling planar graphs. The dependency graph $D(T)$ of a tree T is composed of the dependency graphs of the production instances of T. Clearly, $D(T)$ cannot be planar if one of these subgraphs is non-planar, and it is a simple matter to construct a production with a non-planar dependency graph. For example, the non-planar graph $K_{3,3}$ [5] is the dependency graph of the following production:

$$
\begin{array}{l}
A \rightarrow B \\
\quad B.\,b1 = f(A.\,a1, A.\,a2, A.\,a3) \\
\quad B.\,b2 = g(A.\,a1, A.\,a2, A.\,a3) \\
\quad B.\,b3 = h(A.\,a1, A.\,a2, A.\,a3)
\end{array}
$$



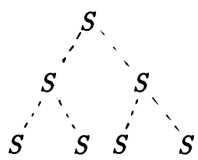
[4] The notation $g(n) = o(f(n))$, which means $\lim_{n \to \infty} sup[g(n)/f(n)] = 0$, signifies that $g(n)$ is asymptotically of lower order than $f(n)$.

[5] The notation $g(n) = \Omega(f(n))$, which means $\lim_{n \to \infty} sup[g(n)/f(n)] > 0$, signifies that $g(n)$ is asymptotically of an order at least as great as $f(n)$.

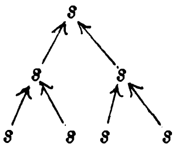
Although for many attribute dependency graphs a distinguished attribute can be evaluated by a method that stores fewer than $O(\log n)$ attribute values, in general a better bound than $O(\log n)$ is impossible, as shown by means of an example. It has been established that pebbling an n-vertex graph that is a complete binary tree with the edges directed towards the root requires $\Omega(\log n)$ pebbles [17]. This establishes a lower bound on the number of attribute values that need to be saved during evaluation, because a derivation tree of the following grammar that is a complete binary tree has a dependency graph that is a complete binary tree with the edges directed towards the root:

$$
\begin{aligned}
S &\rightarrow S\ S \\
&\quad S_1.s = f(S_2.s,\ S_3.s) \\
S &\rightarrow 1 \\
&\quad S.s = 1
\end{aligned}
$$

For example, the tree:

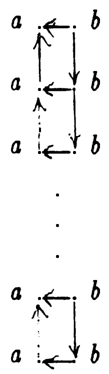


has the dependency graph:



On this example, every evaluation method must use at least $O(\log n)$ attribute values. Consequently, no attribute evaluation method exists that is asymptotically better that the $O(\log n)$ method of Algorithm 7 in terms of the minimum number of attribute values saved during evaluation.

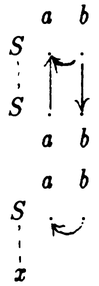
The drawback of the $O(\log n)$-method is that for a certain class of attributed trees, the number of steps required is asymptotically non-linear, for [25] demonstrates the existence of graphs for which using $O(\log n)$ pebbles requires using a non-linear number of steps. In [25], it is shown that for large n, $\Omega(n \log n)$ steps are required to pebble the upper-left vertex of graphs of the form:



when no more than $O(\log n)$ pebbles are used. This establishes that doing attribute evaluation by a method that stores no more than $O(\log n)$ values is inherently nonlinear, because graphs of this form are dependency graphs of the grammar:

$$\begin{aligned} S &\rightarrow S \\ S_1.a &= f(S_1.b,\ S_2.a) \\ S_2.b &= g(S_1.b) \end{aligned}$$

$$\begin{aligned} S &\rightarrow x \\ S.a &= S.b \end{aligned}$$



6. SECONDARY STORAGE AND SEMANTIC CHECKING IN LANGUAGE-BASED EDITORS

Recently, we have worked out methods for using attribute grammars for creating language-based program editors that enforce the syntax and static semantics of a particular language [2, 23, 21]. Each program is represented as a consistently attributed derivation tree. When programs are modified, some of the attributes may no longer have consistent values; incremental analysis is performed by updating attribute values throughout the tree in response to modifications. If an editing operation modifies a program in such a way that context-dependent constraints are violated, the attributes that indicate satisfaction of constraints will receive new values; thus, these attributes can be used to annotate the display in order to provide the user with feedback about the existence of errors.

There is, however, a serious drawback to the use of attribute grammars in language-based editors: the editors use large amounts of storage. This section concerns how to reduce the storage problem by using the techniques developed in this paper to create spill files in secondary storage.

The basic idea is to linearize the unobservable portions of the syntax tree and write them to secondary storage as *spill files*, discarding the values of their attributes. In general, there will be observable attributes remaining, whose values depend on discarded attributes of spilled components. The problem that arises is that, during the course of attribute updating, changes must propagate through the spilled components to such observable attributes.

The attributes of a spilled component are the intermediate values of this calculation; thus, one could rebuild and reevaluate a spilled component when necessary. However, a more efficient alternative is to "compile" the semantic effect of the component at the moment it is removed, by generating an instruction file that can be used to calculate the necessary value without rebuilding the component. To do the calculation, the system interprets the instructions of the file and uses a heap in primary storage to store temporary values.

The idea of creating evaluation-instruction files was originally proposed for non-incremental attribute evaluation in compiler-generating systems [7]. The idea of generating instruction files to compile the semantic effect of a spilled component is a new application of this idea.

In creating an instruction file, the problem of making efficient use of the temporary-value locations is essentially equivalent to the problem treated in this paper – using storage efficiently during on-the-fly evaluation of a distinguished attribute. An algorithm for the latter can be used for the former by reserving a new temporary-value location whenever the evaluation algorithm does an attribute evaluation, and releasing the appropriate location whenever the evaluation algorithm discards a value. For example, topological sorting forms the basis for the following algorithm for generating an instruction file for evaluating attribute b: topologically sort the n attributes in the component to be spilled on which b transitively depends, assign the i^{th} attribute the i^{th} temporary location, and write out instructions to evaluate the attributes in topological order.

To our knowledge, all previously proposed allocation schemes have the property that, in the worst case, the number of storage locations used would be linear in the size of the component to be spilled. In contrast, this paper describes two methods that allow evaluation to be carried out so that the number of stored attribute values is sublinear. To evaluate a distinguished attribute of an n-attribute tree, one method never retains more than $O(\sqrt{n})$ values; the other method never retains more than $O(\log n)$ values. Consequently, if the generation of instructions is based on one of these methods, even in the worst case no more than a sublinear number of temporary-value locations is ever required.

Although the $O(\log n)$-method allows fewer temporaries to be used during evaluation, it may not always be the preferred method for generating instruction files; in the worst case, the number of instructions it generates is nonlinear in the size of the spilled component, whereas the number of instructions generated by the $O(\sqrt{n})$-method is always linear.

ACKNOWLEDGEMENTS

We are grateful for the comments and suggestions of Tim Teitelbaum and Susan Horwitz.

REFERENCES

1. Cook, S.A. An observation on time-storage trade off. *J. Comput. Syst. Sci. 9*, 3 (Dec. 1974), 308-316.

2. Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In Conf. Rec. of the 8th ACM Symp. on Principles of Programming Languages, Williamsburg, Va., Jan. 26-28, 1981, pp. 105-116.

3. Farrow, R.W. Experience with an attribute grammar-based compiler. In Conf. Rec. of the 9th ACM Symp. on Principles of Programming Languages, Albuquerque, N.M., Jan. 25-27, 1982, pp. 95-107.

4. Ganzinger, H. On storage optimization for automatically generated compilers. In *Lecture Notes in Computer Science*, vol. 67: *Theoretical Computer Science, Fourth GI Conference* (Aachen, W. Ger., March 26-28, 1979), K. Weihrauch (ed.), Springer-Verlag, New York, 1979, pp. 132-141.

5. Harary, F. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.

6. Hopcroft, J.E, Paul, W.J., and Valiant, L.G. On time versus space. *J. ACM 24*, 2 (Apr. 1977), 332-337.

7. Jazayeri, M. and Pozefsky, D. Algorithms for efficient evaluation of multi-pass attribute grammars without a parse tree. Tech. Rep. TR 77-001, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, N.C., Feb. 1977 (revised May 1979).

8. Jazayeri, M. and Pozefsky, D. Space-efficient storage management in an attribute grammar evaluator. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct. 1981), 388-404.

9. Kastens, U. Ordered attribute grammars. *Acta Inf. 13*, 3 (1980), 229-256.

10. Kennedy, K. and Ramanathan, J. A deterministic attribute grammar evaluator based on dynamic sequencing. *ACM Trans. Program. Lang. Syst. 1*, 1 (July 1979), 142-160.

11. Kennedy, K. and Warren, S.K. Automatic generation of efficient evaluators for attribute grammars. In Conf. Rec. of the 3rd ACM Symposium on Principles of Programming Languages, Atlanta, Ga., Jan. 19-21, 1976, pp. 32-49.

12. Knuth, D.E. Semantics of context-free languages. *Math. Syst. Theory 2*, 2 (June 1968), 127-145. Correction in *ibid. 5*, 1 (March 1971), 95-96.

13. Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E. Attributed translations. *J. Comput. Syst. Sci. 9*, 3 (Dec. 1974), 279-307.

14. Lindstrom, G. Scanning list structures without stacks or tag bits. *Inf. Proc. Let. 2*, 2 (June 1973), 47-51.

15. Lipton, R.J. and Tarjan, R.E. Applications of a planar separator theorem. In 18th IEEE Symp. on Foundations of Computer Science, Providence, R.I., Oct.-Nov., 1977, pp. 162-169.

16. Lorho, B. Semantic attributes processing in the system DELTA. In *Lecture Notes in Computer Science*, vol. 47: *Methods of Algorithmic Language Implementation*, A. Ershov and C.H.A. Koster (eds.), Springer-Verlag, New York, 1977, pp. 21-40.

17. Paterson, M.S. and Hewitt, C.E. Comparative Schematology. In Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Woods Hole, Mass., June 2-5, 1970, pp. 119-127.

18. Paul, W.J., Tarjan, R.E., and Celoni, J.R. Space bounds for a game on graphs. *Math. Syst. Theory 10*, 3 (1977), 239-251.

19. Pippenger, N. Pebbling. In *Proceedings of the Fifth IBM Symposium on Mathematical Foundations of Computer Science* (Hakone, Japan, May 26-28, 1980), Academic and Scientific Programs, IBM Japan, Tokyo, Japan, 1980.

20. Raiha, K.-J. Dynamic allocation of space for attribute instances in multi-pass evaluators of attribute grammars. In Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colo., Aug. 6-10, 1979. Appeared as: *SIGPLAN Notices* (ACM) *14*, 8 (Aug. 1979), 26-38.

21. Reps, T. *Generating Language-Based Environments.* The M.I.T. Press, Cambridge, Mass., 1984.

22. Reps, T. and Teitelbaum, T. The Synthesizer Generator. In Proc. of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Software Development Environments,

Pittsburgh, Penn., Apr. 23-25, 1984. Appeared as joint issue: *SIGPLAN Notices* (ACM) *19*, 5 (May 1984), and *Soft. Eng. Notes* (ACM) *9*, 3 (May 1984), 42-48.

23. Reps, T., Teitelbaum, T. and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449-477.

24. Schulz, W.A. Semantic analysis and target language synthesis in a translator. Ph.D. dissertation, Univ. of Colorado, Boulder, Colo., 1976.

25. Swamy, S. and Savage, J.E. Space-time tradeoffs for linear recursion. In Conf. Rec. of the 6th ACM Symposium on Principles of Programming Languages, San Antonio, Tex., Jan. 29-31, 1979, pp. 135-142.