Incremental Evaluation of Ordered Attribute Grammars for Asynchronous Subtree Replacements

Josephine Micallef

Columbia University
Department of Computer Science
New York, NY 10027
(212) 280-8178

September, 1988

CUCS-380-88

Abstract

Incremental algorithms for evaluating attribute grammars (AGs) have been extensively studied in recent years, primarily because of their application in language-based environments. Ordered attribute grammars are a subclass of AGs for which efficient evaluators can be constructed. Previous incremental algorithms for ordered attribute grammars only allowed one modification to the program at a time, requiring attribute evaluation due to one change to quiesce before another one due to a second change can start. This article presents new incremental evaluation algorithms for ordered attribute grammars that can handle asynchronous program modifications in an optimal manner. Support for asynchronous changes is necessary in environments for multiple users, where different programmers may be making changes to different parts of the program simultaneously. The key to the optimality of the algorithm is an ordering of the attribute evaluations so that an attribute affected by more than one change will only be evaluated once if the changes happen concurrently.

Copyright © Josephine Micallef

Keywords: Attribute grammars, incremental evaluation, interactive systems, language-based environments, multiuser programming environments, ordered attribute grammars, programming languages.

1. Introduction

Incremental algorithms for evaluation of attribute grammars (AGs) have been the focus of much research in the last few years. These algorithms are of practical importance in language-based environments and incremental compilers based on attribute grammars, where, after a change is made to a program, the attributes affected by the change are evaluated to reestablish consistency among the attributes decorating the program's parse tree.

Incremental attribute evaluators vary along two dimensions. The first dimension determines whether the evaluation strategy is dynamic or static. When evaluating the attributes of a tree T, any evaluator must follow the partial order of T's attribute dependency graph. The dependency graph contains an edge (a, b) between two attributes a and b if a appears in the semantic function defining b. *Dynamic* evaluators maintain the dependency graph at run-time. When a change is made to the program, the dependency graph is updated and attribute evaluations are scheduled by dynamically performing a topological sort on the dependency graph. *Static* evaluators, on the other hand, precompute *plans* that specify the order of evaluation of attributes of each production in the grammar. These plans are created once for each AG during construction of the grammar's evaluator. At run-time, the evaluator determines the order of attribute evaluations using the plans associated with each production instance in T. The advantage of static evaluators is that they are more efficient than dynamic evaluators in both time and space. The disadvantage is that not all well-defined attribute grammars can be evaluated by a static evaluation scheme. However, static evaluators can be constructed for a large subclass of AGs, including most of the ones that arise in practice [12].

The second dimension along which incremental evaluators vary is the model of change used by the algorithm. A change to a program corresponds to a *subtree replacement*, which replaces one subtree in the program's parse tree with another. Some algorithms allow only one change to the program at a time, so that the evaluation from one subtree replacement runs to quiescence before another one starts for a different subtree replacement. Other algorithms handle multiple changes to the program. Some of these require that multiple modifications be synchronized, that is, the evaluator is only started after all modifications have been made. Others allow asynchronous changes, that is, when a change is made, the evaluation of attributes affected by the change starts, but it can be suspended if another change is made affecting attributes that should be evaluated first.

A summary of existing incremental attribute evaluation algorithms according to the classification given above is given in table 1-1. In this article, we present a new static incremental evaluator that can handle multiple asynchronous subtree replacements. We only describe the evaluation algorithm for ordered attribute grammars (OAGs), a large subclass of AGs for which an efficient algorithm for constructing attribution plans is known [5]. However, the same general idea can be used to extend other static tree-walk evaluation strategies (such as the one described in [6]) to handle asynchronous subtree replacements. Our algorithm can be used for the synchronous case as well, and thus fills the two remaining entries (denoted by a star (*)) in the table below.

	Dynamic	*Static*
Single subtree replacement	[9]	[14, 10, 12]
Multiple subtree replacement: Synchronous	[11]	*
Multiple subtree replacement: Asynchronous	[4, 2]	*

Table 1-1: Classification of Incremental Attribute Evaluators

The evaluation algorithm discussed in this article is *optimal* in the following sense: (1) only attributes affected by each modification are evaluated, and (2) an attribute that is affected by more than one subtree replacement still in progress and which has not yet been evaluated in any of them is evaluated once only. In order to accomplish this for the class of OAGs, some run-time checks are required. We define a subclass of OAGs, called the *pairwise ordered attribute grammars* (POAGs), for which this run-time check can be replaced by a table lookup operation, making the evaluator even more efficient.

The rest of the article is organized as follows: Section 2 gives a brief overview of attribute grammars and incremental evaluation. A precise formulation of the problem solved in this article is found in section 3. The incremental evaluation algorithm for OAGs when asynchronous subtree replacements are allowed is presented in section 4. Section 5 defines pairwise ordered attribute grammars, and describes algorithms to construct evaluators for these grammars that record information needed during incremental evaluation. The last section outlines the contributions of this article and compares it to other relevant work.

2. Preliminaries

Attribute grammars were first introduced by Knuth [7] to describe the context-sensitive semantics of a programming language, complementing the way a context-free grammar describes the language's syntax. An AG extends a context-free grammar by attaching *attributes* to the symbols in the grammar, and *semantic equations* defining these attributes to the productions of the grammar. A semantic equation defines an attribute (LHS of equation) as the value of a *semantic function* applied to other attributes of that production (RHS of equation). The attribute on the LHS is *functionally dependent* on the attributes in the RHS of the equation. Attributes are divided into two disjoint classes: *synthesized* and *inherited*. A semantic equation defines a synthesized attribute of the left-hand symbol of a production, or an inherited attribute of one of the right-hand side symbols.

The use of AGs for generating language-based programming environments was originated by Reps [8]. A program is represented by an *attributed derivation tree* (also called a *semantic tree*). The nodes of this derivation tree are labelled with symbols of the grammar. Each node contains fields that correspond to the attributes of its labelling grammar symbol. The value of an attribute instances is computed according to its defining semantic equation. Before an attribute can be evaluated, all other attributes that it is functionally dependent on must have already received values. The functional dependencies among the attributes in the tree create a partial ordering on the attribute instances in the tree. Any attribute evaluation algorithm must obey this partial order, but since the ordering is partial, there may be more than one order of evaluating the attribute instances of the tree.

The program is modified by a sequence of pruning, grafting, or derivation operations on the tree; these operations are collectively called *subtree replacement* operations. After a subtree replacement, the attributes at the root of the replaced subtree may be inconsistent. An attribute is *inconsistent* if its value is not equal to its semantic function applied to the current values of its arguments. An incremental attribute evaluator reevaluates the inconsistent attributes, thus reestablishing consistency among the attributes in the tree.

AG evaluators for both incremental and non-incremental applications fall into two general classes — *dynamic* and *static* evaluators. A dynamic evaluator builds a dependency graph of the attributed tree, where the nodes of the graph are the attribute instances of the tree and the edges correspond to direct and transitive dependencies among the attributes. The nodes of the dependency graph are then topologically sorted, and the attributes evaluated according to their topological order. The disadvantages of a dynamic evaluation strategy are twofold. First, most of the work

is done at runtime. In an incremental editor, this degrades the response time after an edit. Second, in order to build the dependency graph, large structures must be kept around, resulting in an incredible use of storage. Static evaluators overcome both these problems; they are more efficient, both in terms of CPU time as well as memory utilization.

Static evaluators do most of the work once only, during construction of the evaluator. A static evaluator uses a strategy that is pre-computed at construction-time by a static analysis of the grammar. This plan is applicable in any derivation tree of the grammar, and follows the attribute dependencies of the grammar. In the next two sections we describe briefly non-incremental and incremental evaluators for ordered attribute grammars, a subclass of AGs for which static evaluators can be constructed by a polynomial time algorithm.

2.1. Evaluation of Ordered Attribute Grammars

An attribute grammar is ordered if

> ... for each symbol a partial order over the associated attributes can be given, such that in any context of the symbol the attributes are evaluable in an order which includes that partial order [5].

An evaluator for an OAG is guided by plans associated with each production instance in the semantic tree being evaluated. The plan for a production $p: X_0 \rightarrow X_1 \cdots X_n$ is composed of the following basic instructions:

- Eval($X_i.a$) — Evaluates the attribute $X_i.a$ according to the semantic function defining it in production p. $X_i.a$ is a synthesized attribute if $i = 0$ and an inherited attribute if $1 \leq i \leq n$.
- $v(i,k)$ — $\begin{cases} i = 0, & \text{Visits parent of } p \text{ for the } k^{\text{th}} \text{ time.} \\ i > 0, & \text{Visits child } X_i \text{ for the } k^{\text{th}} \text{ time.} \end{cases}$

To evaluate the attributes of a semantic tree T, an evaluator executes the instructions in the plans associated with the production instances of T. Execution starts with the first instruction of the plan for the root production of T. When an *Eval* instruction is encountered, the specified attribute is evaluated, after which the evaluator moves on to the next instruction in the same plan. The plans for two adjoining productions cooperate to evaluate the attributes of an interior node X of the tree T. The inherited attributes of X are evaluated by instructions in the plan for the production where X appears as a right-hand side symbol, while the synthesized attributes of X are evaluated by instructions in the plan for the production where X is on the left-hand side.

If the instruction is a "visit child" (or "visit parent") instruction, then execution is resumed in the plan for the production that applies at the child (or the parent). A function, *MapDown*, keeps track of the next instruction in the plan for the child (or the parent) that should be executed.

A stack implementation of an OAG evaluator is given in Appendix A.

2.2. Incremental Evaluation of Ordered Attribute Grammars

The problem of incremental attribute evaluation can be stated as follows. Starting from a consistently attributed tree T, a subtree S of T is replaced by another tree, S', which is also consistently attributed. Let T' be the tree T with S replaced by S'. The problem is to evaluate the minimum number of attributes in T' so that attribute consistency is reestablished. Optimal solutions to this problem for ordered attribute grammars have been described by Yeh [14] and Reps and Teitelbaum [12]. Here we summarize the algorithm discussed in the latter.

Initially, there are two production instances in T which may have inconsistent attributes. These are the two productions at the point of subtree replacement. If R is the nonterminal occurrence at the root of S' (and necessarily

of S), then the two productions are:

$p: X_0 \rightarrow X_1 \cdots X_m$, *where* $R = X_i$, $1 \le i \le m$, and
$q: R \rightarrow Y_1 \cdots Y_n$

The incremental evaluation algorithm starts executing the first instruction of the plan for production p. Since plans associated with production instances not affected by the subtree replacement do not have to be evaluated, additional information must be maintained to indicate which production instances are affected. This information is stored in the set *Reactivated*, which contains nonterminal occurrences deriving production instances which may have affected attributes. Initially, *Reactivated* contains X_0 and R, which derive the two productions p and q at the point of subtree replacement.

The incremental OAG evaluation algorithm is given in Appendix B. It is similar to the non-incremental version described in the previous subsection, except that the set *Reactivated* is used to limit the scope of attribute evaluations to only those affected. When an attribute a is evaluated, if its value changes and it is an argument in a semantic function defining another attribute b, then the production where b is defined is added to *Reactivated*. "Visit child" and "visit parent" instruction are skipped if the child or the parent are not in *Reactivated*. Otherwise, they are executed in the same way as in the non-incremental algorithm.

3. Problem Formulation

Let T be a parse tree of some ordered attribute grammar G, T' the resulting parse tree after subtree S in T is replaced by S', and T'' the resulting parse tree after subtree R in T' is replaced by R'. The two modifications at S and R are *asynchronous*, that is, the second one may occur while the evaluation of the first one is still in progress. The problem is to design an incremental static evaluator that can handle this scenario in an optimal way, that is, it will only evaluate the minimum number of attributes required to restore consistency.

An incremental evaluator for asynchronous subtree replacements is optimal if it meets the following requirements:

1. For any one modification, the algorithm will evaluate only those attribute instances affected by the modification.
2. For any two (or more) modifications affecting the same attribute a, where both evaluations are still in progress and neither one has yet evaluated a, the algorithm will evaluate a only once.

The second requirement is the more important one for the purposes of this article, so we shall state it a little more formally. Suppose that subtree S was replaced at time t_1, and subtree R at time t_2, where $t_1 < t_2$. Let $AFFECTED_S$ be the set of attributes that were affected (and therefore must be reevaluated) because of the subtree replacement at S, and similarly, $AFFECTED_R$ the set of attributes affected by the subtree replacement at R. Furthermore, suppose that the evaluations from the two modifications overlap, that is,

$$AFFECTED_S \cap AFFECTED_R \neq \emptyset$$

If the evaluation due to the subtree replacement at S is still in progress at the time of the second modification, t_2, then $AFFECTED_S$ can be divided into two subsets: (1) *EVAL*, containing those affected attributes that have already been evaluated at the time of the second replacement, and (2) *UNEVAL*, containing the attributes still needing evaluation.

$$AFFECTED_{S,t_2} = EVAL_{S,t_2} \cup UNEVAL_{S,t_2}$$

Note that all these sets are not known *a priori* but are determined as the evaluation is proceeding. The second optimality requirement states that every attribute a, such that

$a \in UNEVAL_{s,t_2} \cap AFFECTED_R$,
is evaluated only once.

4. Solution for Ordered Attribute Grammars

We first introduce some terminology. During attribute evaluation, we refer to the instruction that is about to be executed as the *current instruction*; the plan containing the current instruction as the *current plan*; and the node deriving the production instance whose associated plan is the current plan as the *current node*. The current node is available in *StackTop.Node*; the current plan in *Plan[StackTop.Node.ProductionIndicator]*; and the index of the current instruction in *StackTop.TableEntry*.

The algorithm consists of three procedures, *StartUp*, *Schedule*, and *Evaluate*, shown in figures 4-1, 4-2, and 4-3 respectively. *Startup* is called whenever a subtree replacement occurs, possibly interrupting another evaluation in progress. *Startup* initializes the state of the evaluation for the new modification and places it on a list of pending evaluations, *PendingList*. This list records the evaluation state of previous subtree replacements whose evaluations have not yet terminated. The pending list is ordered, with the evaluation that will be resumed first at the head of the list. Then, it calls *Schedule*.

```
procedure StartUp(R: nonterminal occurrence at root of replaced subtree)
declare
    p            : production X_0 → X_1 ··· X_m, where R = X_i, 1 ≤ i ≤ m
    q            : production R → Y_1 ··· Y_n
    Reactivated  : set of nonterminal occurrences
    PendingList  : list of evaluations waiting to be restarted, ordered according to which should be restarted first
begin
    Reactivated := {X_0, R}
    push(X_0, MapDown(p,1)) /* evaluation starts at plan for production p which derives R */
    Insert (StackTop, Reactivated) in appropriate place in PendingList
    Schedule()
end
```

Figure 4-1: *StartUp algorithm*

Schedule must determine which evaluation to resume, the one that was previously in progress whose state is recorded in $Stacktop_{current}$ and $Reactivated_{current}$, or the first one in the list of pending evaluations. In order to determine this, it checks whether the current instruction ($Stacktop_{current}$.*TableEntry*) comes before the next instruction to be executed for the first evaluation in the pending list (*PendingList*[1].*Stacktop.TableEntry*) in the computation sequence of the semantic tree representing the program. The *computation sequence* of a semantic tree T is a linearization of the plans associated with the production instances of T, achieved by simulating the operation of an evaluator on T, where instead of executing the instructions, they are appended to the computation sequence. If the current instruction is before the first pending instruction then the current instruction remains the same. If not, then the state of the current evaluation is placed on the pending list and the evaluation at the head of the pending list is made current. The use of the computation sequence to order the pending evaluations is the key to achieving the second optimality requirement stated in section 3.

The rationale behind the operation of the scheduler is that the evaluation that is resumed will eventually reach the other evaluation that was placed on the pending list. This reasoning may be incorrect if a visit to the child or parent

```
procedure Schedule()
declare
    T                       : semantic tree representing program
    Reactivated_current     : set of nonterminal occurrences reactivated by current evaluation
    StackTop_current        : top of stack of current evaluation
    PendingList             : list of evaluations waiting to be restarted, ordered according to which should start first
    Temp                    : holds state of first evaluation of pending list
begin
    if StackTop_current.TableEntry is before PendingList[1].StackTop.TableEntry in computation sequence of T then
        skip
    else
        Remove first element from PendingList and place it in Temp
        Insert (StackTop_current, Reactivated_current) in appropriate place in PendingList
        StackTop_current := Temp.StackTop
        Reactivated_current := Temp.Reactivated
    fi
    Evaluate()
end
```

Figure 4-2: *Schedule algorithm*

that would have reached the other evaluation is skipped because the child or parent were not in *Reactivated*. Therefore *Evaluate* must handle skipped visits in a special way.

Evaluate is responsible for evaluating attributes affected by a modification. It is very similar to the incremental algorithm for single subtree replacements given in section 2.2. The only difference is that if a *visit child* or *visit parent* instruction is about to be skipped because the child or parent is not in *Reactivated*, *Schedule* is called.

4.1. Determining Relative Order Among Plan Instructions

The *Schedule* algorithm described above needed to determine whether an instruction i_1 in plan p_1 occurs before another instruction i_2 in plan p_2 in the computation sequence of T. This can be done as follows. (Step 1) Find the next "visit parent" instruction following i_1 in plan p_1. (Step 2) Simulate the operation of the evaluator to determine the instruction that would be resumed in the parent plan. Repeat these two steps, each time going up to the parent plan, until one of the following happens: (a) instruction j in plan p_2 is encountered, or (b) instruction j in the plan for the root production is encountered. For case (a), if $j < i_2$, then the answer to the question "*Is instruction i_1 executed before instruction i_2?*" is yes, otherwise the answer is no. Case (b) requires some additional work. Repeat steps (1) and (2), but this time for i_2 in plan p_2, until instruction k in the root plan is reached. Then if $j < k$ the answer is yes, else it is no.

4.2. Improvements

The evaluation algorithm given above is asymptotically optimal, but it can still be improved if we can find a more efficient method for determining the relative order among plan instructions, such as precomputing this information at evaluator-construction time. It turns out that this cannot be done for certain OAGs. One such grammar is shown in figure 4-4. Figure 4-5 gives possible attribution plans for the productions in this grammar, such as would be constructed by the algorithm given in [5].

```
procedure Evaluate()
declare
    T: semantic tree representing program
    Reactivated: set of nonterminal occurrences
begin
    repeat
        case StackTop.TableEntry of
            Eval(X.a)    : call semantic function defining X.a
                           increment(StackTop.TableEntry)
                           if NewValue(X.a) ≠ OldValue(X.a) and ∃ attributes that depend on X.a then
                               if X.a is a synthesized attribute then
                                   /* its value can be used in the production where X is on the right hand side */
                                   Reactivated := Reactivated ∪ {X.ParentNode.ProductionIndicator}
                               else /* X.a is an inherited attribute */
                                   /* its value can be used in the production derived from X */
                                   Reactivated := Reactivated ∪ {X.ProductionIndicator}
                               fi
                           fi
            v(i,k), i > 0 : /* descendent visit */
                           increment(StackTop.TableEntry)
                           if X_i ∈ Reactivated then
                               push(StackTop.X_i, MapDown(StackTop.X_i.ProductionIndicator, k))
                           else Schedule()
                           fi
            v(0,k)       : /* ancestor visit */
                           increment(StackTop.TableEntry)
                           if X_0.ParentNode ∈ Reactivated then
                               pop
                           else Schedule()
                           fi
        esac
    until StackIsEmpty or X_0 is root of T or this is the last instruction for plan for production p
end
```

Figure 4-3: *Evaluate algorithm*

production P_0 $\alpha ::= \beta X \gamma$.
attribution
$X.a \leftarrow \ldots$;
$X.c \leftarrow X.b$;
$\ldots$;

production P_1 $X ::= Y$.
attribution
$Y.a \leftarrow X.a$;
$X.b \leftarrow Y.b$;
$Y.c \leftarrow X.c$;
$X.d \leftarrow Y.d$;

production P_2 $Y ::= Z$.
attribution
$Z.a \leftarrow Y.a$;
$Y.b \leftarrow Z.b$;
$Y.d \leftarrow Y.c$;

production P_3 $Y ::= W$.
attribution
$W.a \leftarrow Y.a$;
$Y.b \leftarrow W.b$;
$W.c \leftarrow Y.c$;
$Y.d \leftarrow W.d$;

production P_4 $W ::= Z$.
attribution
$W.b \leftarrow W.a$;
$Z.a \leftarrow W.c$;
$Y.d \leftarrow Z.b$;

production P_5 $Z ::= Q$.
attribution
$Q.a \leftarrow Z.a$;
$Z.b \leftarrow Q.b$;

Figure 4-4: *Attribute Grammer that is not Pairwise Ordered*

The reason that we cannot determine at construction time whether instruction i_1 in plan p_1 is executed before

Evaluate *Y.a*
Move to *Y*
Evaluate *X.b*
Move to parent
Evaluate *Y.c*
Move to *Y*
Evaluate *X.d*
Move to parent

a) Procedure for $X ::= Y$

Evaluate *Z.a*
Move to *Z*
Evaluate *Y.b*
Move to parent
Evaluate *Y.d*
Move to parent

b) Procedure for $Y ::= Z$

Evaluate *W.a*
Move to *W*
Evaluate *Y.b*
Move to parent
Evaluate *W.c*
Move to *W*
Evaluate *Y.d*
Move to parent

c) Procedure for $Y ::= W$

Evaluate *W.b*
Move to parent
Evaluate *Z.a*
Move to *Z*
Evaluate *Y.d*
Move to parent

d) Procedure for $W ::= Z$

Evaluate *Q.a*
Move to *Q*
Evaluate *Z.b*
Move to parent

e) Procedure for $Z ::= Q$

Figure 4-5: *Attribution algorithms for attribute grammar of figure 4-4*

instruction i_2 in plan p_2 is that the answer depends on the structure of the tree containing the two productions p and q associated with the plans p_1 and p_2, respectively. Consider the two attributed trees, T_1 and T_2, shown in figure 4-6 below. Production p is $X ::= Y$ and production q is $Z ::= Q$. If the plan for production q is the current one, and instruction "Evaluate *Q.a*" is being executed, then when the plan for p is eventually resumed, the next instruction is "Evaluate *X.b*" in the case of T_1, whereas in the case of T_2, the next instruction is "Evaluate *X.d*".

In the next section, we define a subclass of OAGs, called the pairwise ordered attribute grammars, for which it is possible to precompute the relative order among plan instructions.

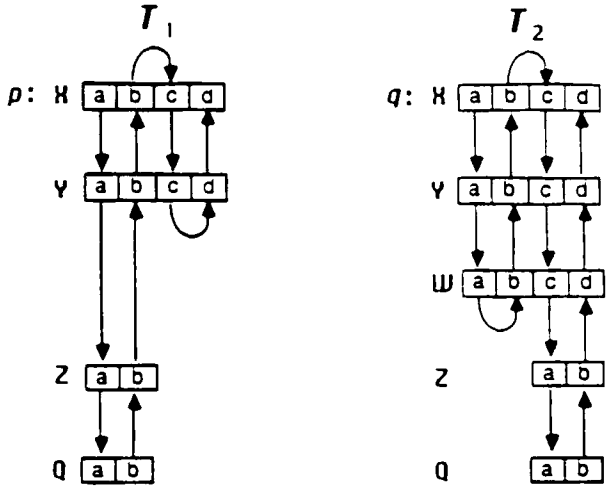

Figure 4-6: *Two semantic trees*

5. Pairwise Ordered Attribute Grammars

Pairwise ordered attribute grammars are defined as a subclass of ordered attribute grammars. An AG is pairwise ordered if:

1. It is ordered, and
2. For each pair of symbols, X and Y, such that $X \stackrel{+}{\rightarrow} Y$, a partial order over the attributes of X and Y can be given, such that in any semantic tree where X is an ancestor of Y, the attributes of X and Y are evaluable in an order which includes that partial order.

5.1. Algorithm to Compute Plans for POAGs

In this section we describe an algorithm that constructs plans for POAGs according to the definition given above. The algorithm is modelled after Kasten's original algorithm to construct visit-sequences for ordered attribute grammars [5]. Only the steps that differ from Kasten's algorithm are described in detail here. Furthermore, we make use of an algorithm to compute transitive dependencies between pairs of symbols in an attribute grammar that was published in [11]. The details of this algorithm are also not repeated below.

In the algorithm below we use the following notation:

- A_X is the set of attributes associated with the nonterminal symbol X. A_X is divided into two disjoint subsets, AI_X, containing the inherited attributes of X, and AS_X, containing the synthesized attributes of X.
- SF is the set of semantic functions associated withthe productions in the grammar. SF_p is the set of semantic functions associated with production p.
- The relation TDS_X contains direct and transitive dependencies between attributes of a nonterminal symbol X.
- The relation TDP_p contains direct and transitive dependencies between attribute occurrences in production p.
- The relation $TDPS_{X,Y}$ contains direct and transitive dependencies between attributes of symbols X and Y, where $X \stackrel{+}{\rightarrow} Y$.

Step 1 and Step 2: Computation of TDS_X and $TDPS_{X,Y}$.

Method: Use algorithms described in the appendix of [11].[1,2] Note that TDP_p is not computed in these first two steps (as is done in [5]) but in step 4. This is done only to simplify the description of the algorithm.

Step 3: Use TDS_X to partition A_X into subsets $A_{X,i}$, $i = 1, \ldots, m$, such that $A_{X,i}$ is a subset of AI_X for odd i and a subset of AS_X for even i. The attributes of X can be evaluated in the order $A_{X,1}, \ldots, A_{X,m}$. The output of this step is a vector *PARTITION* describing the disjoint partitions of A_X.

Method: Same as Step 3 of Kasten's algorithm.

Step 4: Computation of TDP_p.

Method: The algorithm is given in Appendix C. Arcs are added to the (initially empty) TDP_p for the direct dependencies among attribute occurrences in p; the transitive dependencies among attributes of each symbol X in p

---

[1]Our notation follows that of Kastens. It differs from the notation used in [11], where $\overline{DS}(X)$ and $\overline{DP}(X,Y)$ are used instead of TDS_X and $TDPS_{X,Y}$ respectively.

[2]Reps *et al.* use the relation $TDPS_{X,Y}$ in their algorithm to handle multiple *synchronous* subtree replacements.

(given by TDS_X); the transitive dependencies among attributes of the left-hand side symbol X of p and occurrences of each unique symbol Y in the right-hand side of p (given by $TDPS_{X,Y}$); and the dependencies among attributes of each symbol X due to the partitions of X. After adding an edge to TDP_p, other edges required to transitively close TDP are also added. This is accomplished by the function *AddArcTrans* which is the same as defined in [5].

Step 5: Construction of visit-sequences.

Method: Same as Step 5 of Kasten's algorithm.

5.2. Computation of Relative Order Among Plans

For each two productions, $p: X_0 \rightarrow X_1 \cdots X_m$ and $q: Y_0 \rightarrow Y_1 \cdots Y_n$, such that $X_0 \stackrel{+}{\rightarrow} Y_0$, we want to compute:

- Index in *PLAN*[q] where control is transferred after a "Visit child i" instruction in *PLAN*[p], where i is Y_0 or an ancestor of Y_0.
- Index in *PLAN*[p] where control is transferred after a "Visit parent" instruction in *PLAN*[q].

This information will be computed once for each grammar, and stored in the two tables, *MapVisitChildToPlanIndex* and *MapVisitParentToPlanIndex*. *MapVisitChildToPlanIndex*[p,q,i] returns the index of the next instruction in the plan for q to be executed after the "visit child" instruction at index i in the plan for p. *MapVisitParentToPlanIndex*[p,q,i] returns the index of the next instruction in the plan for q to be executed after the "visit parent" instruction in position i in the plan for p.

The first algorithm, shown in figure D-1 in Appendix D, computes the *ANCESTOR* relation for pairs of productions in the grammar, where

ANCESTOR = $\{(p,q) \mid p,q$ *are productions, and* p *is an ancestor of* q *in some parse tree of the grammar*$\}$.

A directed graph G is used, initially containing vertices representing the productions of the grammar and no edges. First, edges are added to G to represent the *PARENT* relation between pairs of productions — an edge between p and q indicates that one of the right hand side symbols of p derives q. The edges added in this step are blue. Then the transitive closure of G is computed to give the *ANCESTOR* relation. Edges added to transitively close G are red. Edge color is used in the next algorithm.

The next algorithm, shown in figure D-2 in Appendix D, builds the two tables *MapVisitChildToPlanIndex* and *MapVisitParentToPlanIndex*. The algorithm sorts the edges (p,q) in the *ANCESTOR* relation in increasing path-of-blue-edges order, that is, first the pairs of productions such that p is the parent of q are considered, (length of path-of-blue-edges is 1), then those such that p is the grandparent of q (length of path-of-blue-edges is 2), and so on. Then, the algorithm iterates over the sorted list of edges, considering them one at time.

If the edge considered, (p,q), is blue (a direct edge), then the actions of the evaluator are simulated to find the instruction in q's plan that is executed after each "visit child" instruction in p's plan, where the child visited is the left hand side symbol of q. If the edge (p,q) is red (a transitive edge), then the principle of dynamic programming is used. We find a production r such that (p,r) and (r,q) are edges in *ANCESTOR*, and (p,r) is a blue edge. The length of the path-of-blue-edges of both (p,r) and (r,q) is less than that of (p,q), and therefore we must have already computed the relative order for these pairs of plans. To find the next instruction that is executed in q's plan after each "visit child" instruction in p's plan, where the child visited is the left hand side symbol of r, we find the first "visit child" instruction in r's plan where the child is an ancestor of the left hand side symbol of q, and then use *MapVisitChildToPlanIndex* to determine where this takes us in q's plan. The entries in the

MapVisitParentToPlanIndex table are computed in a similar way.

6. Contributions and Comparison with Related Work

The primary contributions of this work are:

- A new incremental evaluation algorithm for ordered attribute grammars that can handle asynchronous program modifications in an optimal way.
- The definition of a new subclass of attribute grammars for which the scheduling information for attribute evaluations necessary for asynchronous subtree replacements can be precomputed during construction of the evaluator.

Incremental evaluators that allow asynchronous program modifications are important for environments that support *programming-in-the-many* (PITM), that is, the development and maintenance of large software systems by many different programmers. An incremental evaluation algorithm for multiple asynchronous subtree replacements is used in MERCURY, a generator of language-based environments for PITM [3, 4]. This algorithm does not satisfy our second optimality requirement: an attribute affected by two different subtree replacements may be evaluated twice. Geitz' describes an optimal algorithm for asynchronous subtree replacements which maintains additional information about dependencies between the modified subtrees [2]. His algorithm relies on the computation of $TDPS_{X,Y}$ for each pair of symbols in the grammar, and therefore only works for a subset of AGs.[3] The relation *TDPS* is also used in the algorithm described in [11] for synchronous subtree replacements.[4] The ability to handle synchronous subtree replacement is useful in environments that provide editing commands that do not correspond to subtree replacements, such as transformations, which may result in modifications to more than one part of the tree. All of these algorithms are variants of the optimal incremental evaluator for single subtree replacements described in [9], and are therefore all based on a dynamic evaluation strategy.

The class of ordered attribute grammars was defined by Kastens, who also described polynomial time algorithms for constructing evaluators for them [5]. Yeh describes an incremental version of Kasten's evaluator [14]. The evaluator used in the Cornell Synthesizer Generator for ordered attribute grammars is presented in [12]. This algorithm is also based on Kasten's, and is similar to Yeh's. Both these incremental algorithms only allow single subtree replacements.

Parallel incremental attribute evaluation techniques for ordered attribute grammars are described in [15]. Two versions of parallel evaluation are presented. In the synchronous version, a process is forked for each attribute that is ready for evaluation, i.e., those attributes whose arguments have already been evaluated. In the asynchronous version, a process is forked for any arbitrary attribute evaluation, but this process may have to wait if one of its arguments is not yet available. Zaring's algorithms only apply to single subtree replacement. Boehm and Zwaenepoel also describe a parallel evaluator, but in their case the application area is AG-based compilers, and the algorithm is therefore not incremental [1]. The parse tree is divided into subtrees, which are evaluated in parallel by evaluators executing on different machines. The algorithm uses a combined static and dynamic evaluation strategy: attributes that depend on other attributes associated with nodes in a different subtree are computed dynamically, whereas those whose arguments are in the same subtree are computed statically.

---

[3]This subset is not equal to the class of OAGs or the class of POAGs. It is a subset of the partitionable grammars [13] in the same way that the class of POAGs is a subset of OAGs.

[4]To the author's knowledge, this is the first algorithm to use the *TDPS* relation for scheduling attribute evaluations.

Appendix A. An Evaluator for Ordered Attribute Grammars

Figure A-1 shows a stack implementation of an evaluator for OAGs. *MapDown*(p,k) is a function that returns the next instruction to be executed in the plan for production $p: X_0 \rightarrow X_1 \cdots X_n$ after the k^{th} visit to X_0. For any nonterminal occurrence X in the semantic tree that is being evaluated, the production derived from that nonterminal is found in *X.ProductionIndicator*, and the parent of X is found in *X.ParentNode*.

```
procedure OAGevaluate(root: root of semantic tree to be evaluated)
begin
    push(root, MapDown(root.ProductionIndicator,1))
    repeat
        case StackTop.TableEntry of
            Eval(X.a)      : call semantic function defining X.a
                             increment(StackTop.TableEntry)
            v(i,k), i > 0  : /* descendent visit */
                             increment(StackTop.TableEntry)
                             push(StackTop.X_i, MapDown(StackTop.X_i.ProductionIndicator, k))
            v(0,k)         : /* ancestor visit */
                             pop
        esac
    until StackIsEmpty
end
```

Figure A-1: *Evaluator for Ordered Attribute Grammars*

Appendix B. An Incremental Evaluator for Ordered Attribute Grammars

Figure B-1 is an implementation of the incremental evaluator for OAGs described in section 2.

```
procedure IncOAGevaluate(T: semantic tree; R: nonterminal occurrence at root of replaced subtree)
declare
    p: production X_0 → X_1 ··· X_m, where R = X_i, 1 ≤ i ≤ m
    q: production R → Y_1 ··· Y_n
    Reactivated: set of nonterminal occurrences
begin
    Reactivated := {X_0, R}
    /* start evaluation of plan for production p which derives R */
    push(X_0, MapDown(p,1))
    repeat
        case StackTop.TableEntry of
            Eval(X.a)   : call semantic function defining X.a
                          increment(StackTop.TableEntry)
                          if NewValue(X.a) ≠ OldValue(X.a) and ∃ attributes that depend on X.a then
                              if X.a is a synthesized attribute then
                                  /* its value can be used in the production where X is on the right hand side */
                                  Reactivated := Reactivated ∪ {X.ParentNode.ProductionIndicator}
                              else /* X.a is an inherited attribute */
                                  /* its value can be used in the production derived from X */
                                  Reactivated := Reactivated ∪ {X.ProductionIndicator}
                              fi
                          fi
            v(i,k), i > 0 : /* descendent visit */
                          increment(StackTop.TableEntry)
                          if X_i ∈ Reactivated then
                              push(StackTop.X_i, MapDown(StackTop.X_i.ProductionIndicator, k))
                          fi
            v(0,k)        : /* ancestor visit */
                          increment(StackTop.TableEntry)
                          if X_0.ParentNode ∈ Reactivated then
                              pop
                          fi
        esac
    until StackIsEmpty or X_0 is root of T or this is the last instruction for plan for production p
end
```

Figure B-1: *Incremental Evaluator for Ordered Attribute Grammars*

Appendix C. Computation of TDP_p

The algorithm below computes the relation TDP_p in step 4 of the construction of evaluators for POAGs.

```
procedure step4()
begin
    for each production p: X_0 → X_1 ··· X_k do

        /* add direct dependencies among attribute occurrences in p */
        for each f ∈ SF_p defining X_i.b do
            for each argument X_i.a of f do
                if (X_i.a, X_j.b) ∉ TDP_p then AddArcTrans(TDP_p, (X_i.a, X_j.b)) fi
            od
        od

        /* add transitive dependencies among attributes of each symbol X in p (given by TDS_X) */
        for each unique X_i in p do
            for each edge (c,d) in TDS_{X_i} do
                let (X_i.a, X_i.b) = (c,d) in
                    for each occurrence X_i' of X_i in p do
                        if (X_i'.a, X_i'.b) ∉ TDP_p then AddArcTrans(TDP_p,(X_i'.a, X_i'.b)) fi
                    od
                ni
            od
        od

        /* add transitive dependencies among attributes of each pair of symbols X and Y in p (given by TDPS_{X,Y}) */
        for each X_i in p, 1 ≤ i ≤ k do
            for each edge (c,d) in TDPS_{X_0,X_i} do
                let (X_0.a, X_i.b) = (c,d) in
                    for each occurrence X_i' of X_i in p do
                        if (X_0.a, X_i'.b) ∉ TDP_p then AddArcTrans(TDP_p,(X_0.a, X_i'.b)) fi
                    od
                ni
            od
        od

        /* add dependencies among attributes of each symbol X due to the partitions of X */
        for each nonterminal occurrence X_i' of X_i in p do
            for each X_i'.a do
                for each X_i'.b do
                    if PARTITION[X_i.a] > PARTITION[X_i.b] then[5]
                        AddArcTrans(TDP_p,(X_i.a, X_i.b))
                    fi
                od
            od
        od
    od
end
```

Figure C-1: *Algorithm to compute TDP_p*

[5] Kastens's partitioning algorithm places the attributes that are to be evaluated first in the largest-numbered partition.

Appendix D. Computation of Relative Order Among Plans of POAGs

Figure D-1 shows the algorithm for computing the *ANCESTOR* relation described in section 5.2.

```
procedure Ancestor(out G: a directed graph)
declare
    V       : set of vertices of G
    E       : set of edges of G
    p, q    : productions
    X_i, Y_j : nonterminal symbols
begin
    V := {p | p is a production}
    E := ∅
    for each vertex p: X_0 → X_1 ··· X_m in G do
        for each vertex q: Y_0 → Y_1 ··· Y_n in G do
            if X_i = Y_0, i = 1, ..., m then
                AddBlueEdge(p, q) to G
            fi
        od
    od
    Compute transitive closure of G, adding red edges
end
```

Figure D-1: *Algorithm to compute ANCESTOR relation*

The algorithm in figure D-2 builds the two tables, *MapVisitChildToPlanIndex* and *MapVisitParentToPlanIndex* as described in section 5.2.

```
procedure BuildMaps()
declare
    p                          : production X_0 → X_1 ··· X_m
    q                          : production Y_0 → Y_1 ··· Y_n
    r                          : production Z_0 → Z_1 ··· Z_k
    pIndex, qIndex, rIndex     : integers, used as indices into plans for p, q and r respectively
    EdgeList                   : list of edges
begin
    EdgeList := sort edges (p,q) in ANCESTOR graph in increasing order of length of path of blue edges
                between p and q
    for each edge (p,q) in EdgeList do
        if (p,q) is blue then
            let i be the index of the right hand side (RHS) symbol of p such that X_i = Y_0, i = 1, ...,m in
                qIndex := 1;
                for pIndex := 1 to Length(Plan[p]) do
                    if Plan[p][pIndex] = "Visit Child i" then
                        MapVisitChildToPlanIndex[p,q,pIndex] := qIndex;
                        while Plan[q][qIndex] ≠ "Visit parent" do qIndex := qIndex + 1 od
                        MapVisitParentToPlanIndex[q,p,qIndex] := pIndex + 1;
                        qIndex := qIndex + 1
                    fi
                od
            ni
        else /* (p,q) is red, a transitive edge */
            let r be a production such that (p,r) and (r,q) are edges in ANCESTOR and (p,r) is a blue edge, and
                i be the index of the RHS symbol of p such that X_i = Z_0, i = 1, ...,m and Z_j ±↔ Y_0, j = 1, ...,k, in
                rIndex := qIndex := 1
                for pIndex := 1 to Length(Plan[p]) do
                    if Plan[p][pIndex] = "Visit Child i" then
                        while Plan[r][rIndex] ≠ "Visit child j" do rIndex := rIndex + 1 od
                        MapVisitChildToPlanIndex[p,q,pIndex] := MapVisitChildToPlanIndex[r,q,rIndex]
                        while Plan[q][qIndex] ≠ "Visit parent" do qIndex := qIndex + 1 od
                        rindex := MapVisitParentToPlanIndex[q,r,qIndex]
                        while Plan[r][rIndex] ≠ "Visit parent" do rIndex := rIndex + 1 od
                        MapVisitParentToPlanIndex[q,p,qIndex] := MapVisitParentToPlanIndex[r,p,rIndex]
                        rIndex := rIndex + 1
                    fi
                od
            ni
        fi
    od
end
```

Figure D-2: *Computation of MapVisitChildToPlanIndex and MapVisitParentToPlanIndex*

References

[1] Hans-Juergen Boehm and Willy Zwaenepoel.
Parallel Attribute Grammar Evaluation.
1986.
Rice University.

[2] Bob Geitz.
Asynchronous Subtree Replacement for Language-Based Editors.
1987.
Oberlin College and Cornell University.

[3] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiple-User Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[4] Simon M. Kaplan and Gail E. Kaiser.
Incremental Attribute Evaluation in Distributed Language-Based Environments.
In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 121-130. Calgary, Alberta, Canada, August, 1986.

[5] Uwe Kastens.
Ordered Attribute Grammars.
Acta Informatica 13:229-256, 1980.

[6] K. Kennedy and S.K. Warren.
Automatic Generation of Efficient Evaluators for Attribute Grammars.
In *Third Annual ACM Symposium on Principles of Programming Languages*, pages 32-49. January, 1976.

[7] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[8] Thomas Reps.
Optimal-time Incremental Semantic Analysis for Syntax-directed Editors.
In *Ninth Annual ACM Symposium on Principles of Programming Languages*. January, 1982.

[9] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

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

[11] T. Reps, C. Marceau and T. Teitelbaum.
Remote Attribute Updating for Language-Based Editors.
In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 1-13. St. Petersburg Beach, FL, January, 1986.

[12] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
1988.
Book being prepared for publication.

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

[14] Dashing Yeh.
On Incremental Evaluation of Ordered Attribute Grammars.
BIT 23:308-320, 1983.

[15] Alan Zaring.
Parallel Attribute Evaluation.
1986.
Indiana Universiy and Cornell University.