Efficient Evaluation of Circular Attribute Grammars

LARRY G. JONES
University of Illinois at Urbana-Champaign

We present efficient algorithms for exhaustive and incremental evaluation of circular attributes under any conditions that guarantee finite convergence. The algorithms are derived from those for noncircular attribute grammars by partitioning the underlying attribute dependency graph into its strongly connected components and by ordering the evaluations to follow a topological sort of the resulting directed acyclic graph. The algorithms are efficient in the sense that their worst-case running time is proportional to the cost of computing the fixed points of only those strongly connected components containing affected attributes or attributes directly dependent on affected attributes. When the attribute grammar is noncircular or the specific dependency graph under consideration is acyclic, both algorithms reduce to the standard optimal algorithms for noncircular attribute evaluation.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding—*program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; *syntax*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Attribute circularity, attribute grammars, attribute grammar evaluation, fixed-point computation, incremental attribute evaluation

1. INTRODUCTION

Attribute grammars have been proposed by Knuth [28, 29] as a general method of providing a formal, yet intuitive, framework for specifying the static semantics of programming languages. Attributed translation schemes have proved useful in the specification of compilers used by compiler generation systems (e.g., see [12] and [24]) and also as a basis for the operations required by incremental language-based editors (e.g., [5], [6], [14], and [41]). Attribute grammars have become an important component in the theory of programming language implementation [1, 42].

Knuth has shown that a proper subclass of attribute grammars, specifically those having only *synthetic* attributes, is powerful enough to express the static semantics of any derivation tree. On the other hand, inclusion of *inherited*

This work was supported in part by National Science Foundation grant MIP-87-10865.
Author's address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

attributes often simplifies the descriptions of many properties of programming languages. Attribute grammars having both synthetic and inherited attributes may not be *noncircular*, a condition suggested in [28] that guarantees the existence and uniqueness of a consistent assignment of attribute values. Knuth [29] gives an algorithm for determining when an attribute grammar is noncircular. However, his algorithm has worst-case running time that is exponential in the size of the attribute grammar. It has since been shown that testing an arbitrary attribute grammar for the noncircularity condition is an inherently intractable problem [17]. Nevertheless, there is much empirical evidence [32] that suggests that the noncircularity property can be established in polynomial time for most (noncircular) attribute grammars that arise in practice. In fact, most attribute grammars actually used by compilers or language-based editors are members of restricted classes of attribute grammars for which noncircularity is guaranteed [7, 23, 25, 30].

We note that the noncircularity condition for attribute grammars is not a necessary one; it suffices that all circularly defined attributes have a fixed point obtainable with finite (and efficient) computation. Recently, interest has risen in the use and evaluation of circular attribute grammars for simplifying the description of semantics that are inherently circular. For example, in [20] and [21], attribute techniques are used to describe properties of circuits in a hierarchical VLSI design system, the design hierarchy serving a similar function to that of a derivation tree. As most nontrivial VLSI circuits have feedback loops, many useful attributes are circularly defined. Circular attribute grammars have also been useful in the context of data-flow analysis and code optimization [3, 13, 38]. More recently, circular attributes have been used to describe run-time semantics as a basis for an environment that supports the incremental execution of programs [43].

In considering the use of circular attribute grammars, two questions arise: (1) How can we ensure the (circular) attributes are computable in finite time and (2) how can we efficiently evaluate circular attributes? For the answer to (1), we note that the computation of attributes in any graph can be viewed as a fixed-point computation over a system of simultaneous equations. It is well known that, under certain circumstances, such systems have a unique least fixed point obtainable with finite computation. For example, if the equations are monotone and yield values defined over a lattice of finite height, an iterative refinement approach may be used. In another example, if the system forms a set of linear equations over the rationals, Guassian elimination may be used.

As a solution to (2), we present in this paper efficient algorithms for exhaustive and incremental evaluation of circular attributes under any conditions that guarantee finite convergence. The algorithms presented here are derived from those for noncircular attribute grammars [27, 28, 34] by partitioning the underlying attribute dependency graph into its strongly connected components and ordering the evaluations to follow a topological sort of the resulting directed acyclic graph. The algorithms are efficient in the sense that their worst-case running time is proportional to the cost of computing the fixed points of only those strongly connected components containing affected attributes or attributes directly dependent on affected attributes. When the attribute grammar is

noncircular or the specific dependency graph under consideration is acyclic, both algorithms reduce to the standard optimal algorithms for noncircular attribute evaluation presented in [27], [28], and [34].

As we have mentioned, testing an attribute grammar for noncircularity is a well-known intractable problem. Noncircular attribute grammars that arise in practice are usually members of strict subclasses of attribute grammars for which noncircularity is easily established. The algorithms and concepts presented in this paper extend the usable set of attribute grammars beyond the noncircular ones and provide a general technique for circumventing circularity testing.

1.1 Related Work

Babich and Jazayeri [3, 4] studied circular attribute *reevaluation* in the context of code optimization. They presented exhaustive evaluation algorithms that evaluate attributes during multiple traversals of the derivation tree. The attributes presented are guaranteed to converge after two traversals.

Skedzeleski examined circular *time-varying attributes* and their exhaustive reevaluation [38]. He noted that the evaluation of general circular attributes may not terminate, or if it does, the solution arrived at may be dependent on the initial conditions and evaluation order. He gave several examples of circular attributes solving useful problems in the area of data-flow analysis (e.g., constant propagation and live variable analysis) and that satisfy conditions that guarantee convergence to a unique solution. The exhaustive evaluation strategy he gave was based on change propagation: An attribute is reevaluated only after one of its arguments has changed value. Reps has since pointed out that change propagation by itself is inadequate to prevent spurious evaluations [33].

Ganzinger, Giegerich, Möncke, and Wilhelm have described a compiler generation system that allows the user to specify explicitly circular attributes and to have some control over the evaluation order in an exhaustive attribute evaluator [15].

Efficient algorithms for both the exhaustive and incremental evaluation of circular attribute grammars were presented by Jones and Simon [21] in the context of hierarchical VLSI design systems based on attribute grammar techniques. The techniques suggested are general in the sense that they may be used under any conditions guaranteeing finite convergence to a least fixed point. The algorithms described in this paper are based on those suggested in [21] and include several improvements. As an example of a set of conditions that guarantee finite convergence, the authors suggested that all circularly defined attributes have semantic functions that are monotone and yield values from a domain forming a lattice of finite height.

Farrow [13] has suggested similar techniques for the exhaustive evaluation of circular attributes. As a condition guaranteeing finite convergence, he restricts the semantic functions of circularly defined attributes to be monotone, to yield values from domains forming complete partial orderings, and to satisfy the *ascending chain condition.* In comparison with [21], the ascending chain condition (as it is defined in [13]) is equivalent to requiring a finite bound on the height of the domain of the circular attributes. As any lattice is a complete partially

ordered set, the conditions of [13] are more general than those suggested in [21]. However, it is well known that any partially ordered set can be embedded in a complete lattice [39], and so this difference is slight.

The use of attribute grammars for the specification and construction of language-based editors and the notion of incremental evaluation of attributes was first introduced by Demers, Reps, and Teitelbaum in [9] as the basis for the *Synthesizer Generator* [35]. The Synthesizer Generator automatically generates language-based editors from an attribute grammar description of the target language. The systems generated are based on the incremental attribute evaluation algorithms presented in [33], [34], and [37] with the exception that whenever the attribute grammar is an *ordered attribute grammar* [23] the more efficient algorithm of [44] is selected. Other systems utilizing incremental attribute evaluation techniques include the *Pascal Oriented Editor* (*POE*) [14], the *Software Automation, Generation, and Administration* (*SAGA*) project [6], the *Incremental Circuit Editor* (*ICE*) system [20], and the *Program System Generator* (*PSG*) system [5].

Optimal incremental evaluation of general noncircular attribute grammars was first presented by Reps in [33] and is discussed further in [34] and [37]. The incremental algorithm presented in this paper is largely based on the approaches taken by Reps but with extensions to handle circular attribute grammars. Optimal incremental algorithms for subclasses of noncircular attribute grammars have been investigated by Yeh [44], Reps [34], Johnson and Fischer [19], and Alblas [2]. Nonoptimal incremental evaluation methods used to introduce nonlocal dependencies and to reduce overhead have been investigated by Johnson and Fischer [18], Demers, Rogers, and Zadeck [10], Hoover [16], and Reps, Marceau, and Teitelbaum [36]. In particular, Hoover's method [16], although not optimal, usually exhibits run-time performance equal to the best incremental evaluation methods by approximating a topological sort.

1.2 Definitions

An *attribute grammar* is a context-free grammar with a set of *attributes* associated with each of the symbols in the grammar. Each production rule of the grammar has an associated set of *semantic equations*. A semantic equation expresses the value of an attribute attached to a symbol appearing in the production rule by a *semantic function* applied to the values of other attributes appearing in the production. The attributes whose values are used by the semantic function are known as the *arguments* of the attribute whose value is derived.

Given a derivation tree constructible from the context-free grammar, we can obtain a corresponding *attributed derivation tree* by attaching distinct *instances* of each attribute of a grammar symbol, X, to every node in the derivation tree with label X. We will typically drop the use of "instance" when it is clear from the context that we are dealing with the formal definition of an attribute in the attribute grammar or an attribute instance appearing in the attributed derivation tree. The *value* of an attribute instance of a treenode t is defined by the appropriate semantic equation corresponding either to the production rule used in deriving t or to the rule used in deriving the children of t. The arguments of

an attribute of t are therefore instances of attributes in treenodes local to t, specifically, the parent of t, the siblings of t, the children of t, or t itself.

The set of attributes associated with a given grammar symbol, X, can be partitioned into two disjoint sets: the set of *synthetic* attributes of X and the set of *inherited* attributes of X. Each semantic equation corresponding to a production rule either defines a synthetic attribute of the grammar symbol appearing on the left-hand side of the production or defines an inherited attribute of a grammar symbol appearing on the right-hand side of the production. Intuitively, synthetic attributes pass semantic information up toward the root of the derivation tree, whereas inherited attributes pass semantic information down toward the leaves of the tree.

An attribute instance is said to be *consistent* if its value is equal to the application of the corresponding semantic equation to the values of its arguments. It is said to be *globally consistent* if it is consistent and every attribute instance that it is directly or indirectly dependent on is consistent. An attributed derivation tree is said to be consistent if all attribute instances are consistent (equivalently, if all attribute instances are globally consistent).

The *meaning* of an attributed derivation tree is considered to be expressed by the values of the attributes when the tree is consistent. It is convenient to view the problem of determining the meaning of an attributed derivation tree as one of solving the underlying system of simultaneous semantic equations for the values of the attribute instances, obtaining consistent assignments for all attributes. An *attribute evaluation scheme* is an algorithm for obtaining such a globally consistent assignment.

Many attribute evaluation schemes have been proposed in the literature. *Exhaustive evaluation* methods compute the values of all attributes in the derivation tree and are appropriate in the context of batch program compilation. *Incremental evaluation* methods compute new values for attributes that were made inconsistent by modifications to a consistent attributed derivation tree. Incremental evaluation is appropriate in the context of systems such as incremental compilers, language-based programming environments, or incremental computer-aided engineering systems.

Static evaluation methods determine a priori an ordering among the attributes of a fixed attribute grammar that leads to an efficient evaluation order of the attribute instances of any possible derivation tree. *Dynamic evaluation* methods determine an efficient ordering among the attribute instances *during* evaluation. Static methods often show superior run-time performance when compared to dynamic methods, since the attributes are ordered during system generation and enforcing this ordering can be done with low overhead during attribute evaluation. Static methods are not always appropriate: Given an arbitrary attribute grammar, it may not be possible to establish an ordering among the attributes that reflects the ordering of attribute instances implied by any derivable attributed derivation tree. In this paper we present *dynamic* attribute evaluation algorithms for both *exhaustive* and *incremental* evaluation of a general class of attribute grammars. This class covers the entire set of well-defined noncircular attribute grammars and includes all circular attribute grammars for which a consistent assignment can be found in finite time.

1.3 Attribute Circularity

A natural problem that arises when using attribute grammars to specify properties of a programming language is determining the existence of a consistent assignment to all the attributes of any derivable attributed derivation tree. If we assume that semantic functions have no side effects (they are equations) and that the domain of each semantic function covers the range of values its arguments can assume, then the question is, under what conditions are these equations solvable and when is the solution unique?

An attribute instance is said to be *circularly defined* if its value is directly or indirectly dependent on itself. If there are no circularly defined attribute instances, there must be a unique solution to the set of semantic equations. It is therefore useful to determine a priori whether a given attribute grammar may lead to circular definitions in the attributed derivation trees constructible from that grammar.

We may recognize attribute circularity in an attributed derivation tree by examining a graph that exhibits the underlying dependency structure of the attribute instances. Given an attributed derivation tree, T, we construct a directed graph, $D(T)$, called the *dependency graph* of T. $D(T)$ contains a vertex corresponding to every attribute instance in T. A directed edge from x to y is in $D(T)$ if and only if (iff) the attribute instance corresponding to x is an argument of the attribute instance corresponding to y. An attributed derivation tree, T, contains a circularly defined attribute instance iff there is a directed cycle in $D(T)$ containing the vertex corresponding to this instance. An attribute grammar is said to be *circular* if it is possible to construct an attributed derivation tree from the grammar that has a cycle in its underlying dependency graph. If no such cycle appears in the dependency graph of any possible attributed derivation tree, then the attribute grammar is said to be *noncircular*.

In general, the family of derivation trees constructible from any nontrivial context-free grammar is infinite. Knuth [28, 29] showed that, given an attribute grammar, it is possible to construct a finite collection of directed graphs for each production such that every derivation tree in the language has an acyclic dependency graph iff none of the constructed graphs contains a directed cycle. Although this shows that noncircularity is a decidable property of attribute grammars, Knuth's algorithm has worst-case running time that is exponential in the size of the attribute grammar. Jazayeri, Ogden, and Rounds [17] have shown that noncircularity testing is an inherently intractable problem and that *any* deterministic algorithm will exhibit exponential run-time behavior for an infinite number of attribute grammars. Räihä and Saarinen [32] revealed that the complexity is dominated by the maximum number of graphs associated with any one nonterminal during the noncircularity test. They conjectured that for most practical attribute grammars this number is small and noncircularity testing is feasible. There is much empirical evidence to support this conjecture, and a number of researchers have developed methods for lowering the overhead associated with noncircularity testing [8, 11, 22, 31].

While noncircularity is a *sufficient* condition to guarantee the existence and uniqueness of a solution, it is not a *necessary* condition. The evaluation of attributes having an arbitrary dependency graph can be viewed as a fixed-point

computation over the system of semantic equations. It is well known that, under certain circumstances, such systems of simultaneous equations have a unique least fixed point obtainable with finite computation. For example, nonsingular systems of linear equations over the reals can be solved using Gaussian elimination, or if the equations represent monotone functions over a lattice of finite height, the solution can be reached by an iterative refinement approach.

Many properties of programming languages are naturally defined using circular attribute grammars; for example, problems from the area of data-flow analysis often involve iterative solutions and are solved using classical fixed-point techniques [26]. A number of circular attribute grammars for expressing data-flow problems have been proposed in the literature [3, 13, 38]. Circular attribute techniques are also useful as a basis for the implementation of VLSI design automation and computer-aided engineering systems [20, 21].

It is not always necessary to require the existence or uniqueness of a consistent assignment. For example, problems in data-flow analysis for code optimization may often have more than one solution and are solved for *some* fixed point, but not necessarily for the *least* fixed point. In another example, when attributes are used to express run-time semantics of programming languages or circuits, it is entirely possible that there is no solution to the system of equations. Such is the case when a user's program contains an infinite loop [43] or when a circuit contains signals that oscillate indefinitely [20].

2. EXHAUSTIVE EVALUATION OF CIRCULAR ATTRIBUTE GRAMMARS

Given an attributed derivation tree, T, the problem of exhaustive attribute evaluation is to determine assignments of all attribute instances such that T is consistent. An edge (x, y) in the dependency graph $D(T)$ implies that the value of attribute y is dependent on the value of attribute x (x is an argument of y). If the attribute grammar is noncircular, $D(T)$ is acyclic, and the value of x is independent (directly or indirectly) of the value of y. It is therefore natural to order the evaluations so that x is evaluated before y, avoiding excessive evaluations. If we evaluate attributes following the partial ordering implied by the dependency graph, then no attribute is evaluated before any of its arguments, and each attribute is evaluated exactly once. This ordering is called a *topological sort* [27] of the dependency graph and forms the basis for the optimal algorithm for the exhaustive evaluation of noncircular attribute grammars as suggested in [28].

The topological sort method is not directly applicable to circular attribute grammars, since the dependency graph underlying an attributed derivation tree of a circular attribute grammar may contain cycles and no such partial ordering is obtainable. It is possible, however, that not all attribute instances are interdependent and that an ordering among the equivalence classes of attributes may be obtained by carefully partitioning the attributes. Restricting fixed-point computations to each partition and arranging these computations to follow the implied partial ordering will yield a strategy that is, in general, more efficient than the naive one.

Let T be an attributed derivation tree and $D(T)$ the associated dependency graph. Let $C(T)$ be the *SCC graph* of $D(T)$; that is, $C(T)$ is the directed acyclic

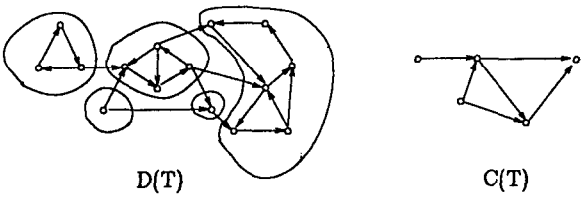

Fig. 1. A dependency graph and its SCC graph.

graph that results when each strongly connected component of $D(T)$ is contracted into a single node. In Figure 1 we present an example of a dependency graph containing cycles and its corresponding SCC graph. There is an edge (b, c) in $C(T)$ iff b and c are two distinct strongly connected components of $D(T)$, and there are attributes x and y in b and c, respectively, such that there is an edge (x, y) in $D(T)$. It follows that an edge (b, c) in $C(T)$ represents a functional dependence of c's fixed point on b's fixed point. Since b's fixed point is in no way dependent on c's fixed point, b should be evaluated first. If two attributes are dependent on each other's fixed-point value, then they are part of the same strongly connected component of $D(T)$, and we evaluate them as a unit, in a single fixed-point computation. The algorithm presented in Figure 2 evaluates the attributes of T by scheduling the fixed-point computations of the strongly connected components of $D(T)$ to follow a topological sort of the partial ordering formed by $C(T)$.

After the initial construction of the graph $C(T)$, the algorithm maintains the invariant that the set S contains all vertices of $C(T)$ with indegree zero. During each iteration of the loop, exactly one strongly connected component, say, b, is selected and removed from S. The fixed point of b is computed, and all edges in $C(T)$ from b to any of its successors are deleted. If this deletion causes the indegree of some successor c to become zero, then c is inserted into S.

Initially, vertices in $C(T)$ with indegree zero represent strongly connected components of $D(T)$ containing only attributes whose values are independent of the values of any attributes outside the strongly connected component. Following a fixed-point computation of such strongly connected components, each member attribute will be globally consistent. By induction on the longest path in the initial graph $C(T)$ from a vertex with indegree zero to a particular vertex representing a strongly connected component b of $D(T)$, all attributes not in b that serve as arguments of attributes in b will be globally consistent at the time b is selected for fixed-point computation. It follows that after any fixed-point computation all member attributes of the strongly connected component will be globally consistent. By delaying the fixed-point computation of a strongly connected component until all its arguments are globally consistent, we guarantee that only one fixed-point computation is required for each strongly connected component.

At each iteration, exactly one strongly connected component is evaluated, deleted from S, and never reinserted. The number of iterations is therefore bounded by the number of strongly connected components in $D(T)$. Since $C(T)$ is always a directed acyclic graph (edge deletion preserves this property), if it

```
let C(T) be the SCC graph of D(T);
insert into S all vertices of C(T) with indegree zero;
while S is not empty do
  select and remove a vertex b from S;
  compute the fixed point of b;
  for each successor, c, of b in C(T) do
    remove (b,c) from C(T);
    if indegree(c) is zero then
      insert c into S;
    fi
  od
od
```

Fig. 2. Efficient exhaustive evaluation of circular attributes.

contains any edges then there must be at least one vertex having indegree zero whose fixed point has not been computed and is therefore a member of S. At loop termination, S is empty, $C(T)$ contains no edges, and the fixed points of all strongly connected components have been computed.

The SCC graph of a directed graph, $G = (V, E)$, can be constructed in $O(|E| + |V|)$ time using depth-first search [40]. Because of the bounded degree of $D(T)$, $C(T)$ may be constructed in $O(n)$ time for n attributes in $D(T)$. Let $|F_i|$ denote the cost of computing the fixed point of the ith strongly connected component of $D(T)$, for N strongly connected components. Since, in exhaustive evaluation, every attribute must be evaluated, the cost of computing the fixed points of all strongly connected components is at least linear in the number of attributes. The algorithm has worst-case running time bounded by $O(\sum_{i=1}^{N} |F_i|)$. In the special case that $D(T)$ is acyclic, the algorithm reduces to topological sort, which is the standard optimal algorithm presented by Knuth [27, 28] for the exhaustive evaluation of noncircular attribute grammars.

3. COMPUTING THE FIXED POINT OF A STRONGLY CONNECTED COMPONENT

In the previous section, we gave an efficient algorithm for the exhaustive evaluation of circular (and noncircular) attribute grammars. We showed how the attribute instances could be partitioned into interdependent sets by identifying the strongly connected components of the dependency graph, $D(T)$. A topological ordering of the attribute sets based on the SCC graph, $C(T)$, constructed from $D(T)$ yields an order guaranteeing that each strongly connected component (and therefore each attribute) is involved in exactly one fixed-point computation. Although the cost of an individual fixed-point computation is dependent on both the size of the partition and the method used, it is anticipated that this cost will normally be small compared to the cost of computing the fixed point of all attributes in the absence of partitioning.

The evaluation methods presented in this paper are general in the sense that they can be used with any technique for computing the fixed points of the strongly connected components. For completeness, in this section we give an efficient algorithm for the computation of the fixed points of the strongly connected components for a proper subset of the set of all circular attribute grammars that yield only systems of semantic equations that have a finitely

computable fixed point. Following [21], we require that the domain of any attribute instance occurring in a nontrivial strongly connected component be a lattice of finite height and that the semantic function f of such an attribute be monotone.

We denote the domain of an attribute instance x as $\mathbf{D}(x)$. Let $X \equiv (x_1, x_2, \ldots, x_n)$ be an ordered set of all the attribute instances occurring in an attributed derivation tree T. Then the domain for tuples in X is $\mathbf{D}(X) \equiv \mathbf{D}(x_1) \times \mathbf{D}(x_2) \times \cdots \times \mathbf{D}(x_n)$. Let f_x be the semantic function corresponding to some attribute instance x. The set of arguments of x may be a small subset of all the attribute instances; however, it is convenient to view f_x as a function of all attribute instances of T, even though many of these arguments are ignored. Then f_x is a mapping f_x: $\mathbf{D}(X) \rightarrow \mathbf{D}(x)$. Let $F_X \equiv (f_{x_1}, f_{x_2}, \ldots, f_{x_n})$ be the ordered set of semantic functions corresponding to the attribute instances of X. F_X may be viewed as a mapping, F_X: $\mathbf{D}(X) \rightarrow \mathbf{D}(X)$, which derives new values for the ordered set of attribute instances X by applying the corresponding semantic functions to a tuple of attribute instances. Then T is consistent whenever $X = F_X(X)$, that is, whenever X is a fixed point of F_X.

Let b be a strongly connected component of $D(T)$. In the evaluation method presented in the previous section, at the time the computation of b is scheduled, all attribute instances that b is dependent on are globally consistent and may be considered constant. It follows that, at the time b is scheduled for evaluation, any attribute instance not in b that occurs in the semantic function of an attribute in b can be replaced by its globally consistent value, and the attribute instances in b can be expressed by modified semantic functions having only attributes of b as arguments. We may then construct an ordered set, $Y \equiv (x_{i_1}, x_{i_2}, \ldots, x_{i_k}) \subseteq X$, for k attribute instances in b, and a corresponding ordered set of the modified semantic functions, $F_Y \equiv (f'x_{i_1}, f'x_{i_2}, \ldots, f'x_{i_k})$. The equation $Y = F_Y(Y)$ represents a restricted fixed-point computation that may be used to find a consistent assignment for the attribute instances of the strongly connected component b. We note that each attribute instance in the restricted set Y and its corresponding semantic function in F_Y obeys our requirements for guaranteeing finite computation; that is, the domain of each attribute instance in Y is a lattice of finite height, and its semantic function in F_Y is monotone. Furthermore, $\mathbf{D}(Y)$ forms a lattice of finite height, and F_Y is monotone.

Shown in Figure 3 is a simple method for computing the least fixed point, $Y = F_Y(Y)$, of the strongly connected component b. Let $\perp_x$ denote the least element in $\mathbf{D}(x)$. Let $\perp_Y$ denote the least element in $\mathbf{D}(Y)$, that is, the tuple $(\perp_{x_{i_1}}, \perp_{x_{i_2}}, \ldots, \perp_{x_{i_k}})$. Since the operator F_Y is monotone yielding values over a lattice of finite height, the procedure will converge on a fixed point after finitely many iterations. As $\perp_Y$ is the least element of $\mathbf{D}(Y)$, the fixed point reached will be the least fixed point.

In each iteration of the algorithm, every attribute in Y is evaluated. For k attributes in b, each iteration performs k evaluations. The iterations continue as long as there is at least one attribute in Y whose value has changed. In the worst case, exactly one attribute could increase in value during each iteration. Let Y be the least member of $\mathbf{D}(Y)$ satisfying the equation $Y = F_Y(Y)$. For $x_i \in Y$, let h_{x_i} represent the height of x_i in $\mathbf{D}(x_i)$. Then $\sum_{i=1}^{k} h_{x_i}$ iterations may be necessary

```
i ← 0;
Y^0 ← ⊥_Y;
repeat
    i ← i+1;
    Y^i ← F_Y(Y^{i-1});
until Y^i=Y^{i-1};
```

Fig. 3. Simple method for computing the least fixed point $Y = F_Y(Y)$.

in the worst case, and the algorithm performs $O(k \sum_{i=1}^{k} h_{x_i})$ attribute evaluations in the worst case.

3.1 Efficient Fixed-Point Computation

In the above algorithm, throughout iteration i, all argument values used in deriving a new value for an attribute instance of Y were derived during iteration $i - 1$. We may relax this restriction and allow an attribute's value to be derived from argument values that were computed during the same iteration. Now, however, the order in which the attributes are evaluated can significantly impact the number of iterations necessary to reach the fixed point. For example, if the attributes are processed in reverse order, with respect to $D(T)$, then there may be little or no advantage to the relaxation technique. Since there is no topological ordering of the attributes of a strongly connected component, we order the evaluations of the attributes in b to follow a depth-first traversal of the attributes in b with respect to $D(T)$. Finally, there is little point in reevaluating an attribute if none of its arguments have changed value since the last evaluation of the attribute.

We present in Figure 4 an algorithm for computing the fixed point of a strongly connected component that is based on the above observations. Initially, we start with every attribute in the strongly connected component having value $\perp_x$, the least element of the attributes domain. We construct a scheduling set, s_b, which, between successive iterations, contains all possibly inconsistent attributes in b. Initially, s_b will be exactly the set of all attributes in b. We evaluate the attributes of b in depth-first order commencing with the attributes in s_b. The two sets s and s_b differentiate the rounds of evaluations. The set s may be considered as a working set of roots of the spanning forest, while the set s_b is the set of newly discovered, possibly inconsistent attributes that will be used as roots during the next round of evaluations.

The depth-first evaluation of the attributes of a strongly connected component is shown in Figure 5. As we visit each attribute, x, we mark and evaluate it. If x's value has not changed, then the subtree rooted at x is unaffected by the new value and need not be traversed. Suppose the value of x has changed, and let y be a successor of x. If y is marked, then we know that it is already in the spanning forest and has been assigned a value that may be inconsistent with the new value of x. We schedule y for reevaluation in the next iteration by inserting it into s_b. If y is not marked, then it is not in the spanning forest, and we visit it.

In the beginning, all attributes in b are members of s_b and will therefore be evaluated and made consistent (even if only temporarily). Suppose that a consistent attribute y is made inconsistent. Then y has some argument x that has changed value. The attribute y will be reevaluated, since if it is marked it will be entered into s_b and if it is unmarked it will be reevaluated directly.

```
initially s_b is the set of all attribute instances in b;
initialize each attribute instance, x ∈ s_b, to ⊥_x;
initially all attributes in b are considered unmarked;

while s_b is not empty do
  s ← s_b;
  s_b ← φ;
  while s is not empty do
    select and remove an attribute x from s;
    if x is not marked then
      TREE_EVALUATE(x) {i.e. evaluate the spanning tree rooted at x};
    fi
  od
  clear marked attributes;
od
```

Fig. 4. Fixed-point computation of a strongly connected component, b.

```
TREE_EVALUATE(x):
  mark x as being visited;
  evaluate x;
  if the new value of x is not equal to its old value then
    for each successor, y, of x in b do
      if y is not marked then
        TREE_EVALUATE(y);
      else
        insert y into s_b;
      fi
    od
  fi
end TREE_EVALUATE;
```

Fig. 5. Depth-first evaluation of a spanning tree of b commencing with x.

After the first iteration of the outermost loop, an attribute x is evaluated only if at least one of its arguments has changed. If we charge each argument attribute with any evaluations it may have caused following a change in value, because of the bounded outdegree of $D(T)$, an attribute x can be charged with at most $O(h_x)$ evaluations. It follows that the algorithm we have presented performs $O(\sum_{i=1}^{k} h_{x_i})$ attribute evaluations in the worst case. In the case where the h_{x_i} are bounded by a constant, the number of attribute evaluations performed is linear in the size of the strongly connected component.

3.2 An Example

In Figure 6 we give definitions of circular attributes that solve the *constant propagation* problem for a subset of the grammar rules describing a hypothetical language. The goal in constant propagation is to determine at each point in the program the set of variables having specific constant values. Several code optimization techniques may then be applied in an effort to reduce the run time of the program. For example, expressions always yielding specific constant values

$S' ::= S$
(1) $S'.in \leftarrow \bot$;
(2) $S.in \leftarrow S'.in$;
(3) $S'.out \leftarrow S.out$;

$S ::= \textbf{while}\ expr\ \textbf{do}\ S_1\ \textbf{od}$
(4) $S_1.in \leftarrow S.in \sqcup S.out$;
(5) $S.out \leftarrow S_1.out$;

$S ::= \textbf{if}\ expr\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
(6) $S_1.in \leftarrow S.in$;
(7) $S_2.in \leftarrow S.in$;
(8) $S.out \leftarrow S_1.out \sqcup S_2.out$;

$S ::= S_1 ; S_2$
(9) $S_1.in \leftarrow S.in$;
(10) $S_2.in \leftarrow S_1.out$;
(11) $S.out \leftarrow S_2.out$;

$S ::= \textbf{id} \leftarrow expr$
(12) $S.out \leftarrow gen(S.in, \textbf{id}, expr) \sqcup kill(S.in, \textbf{id})$;

Fig. 6. Circular attribute grammar for constant propagation.

may be eliminated by replacing them with references to the constants; similarly, variables having constant values may be eliminated, loops may be unrolled, and so on.

Kildall has presented a generalized and formal technique for solving many common code optimization problems that is based on obtaining a least fixed point of a set of simultaneous equations underlying a control flow graph of the program [26]. Among other problems, he shows how this method can be used to solve the constant propagation problem. Circular attributes that express the constant propagation problem and that are based on Kildall's method have also been presented in [38]. For clarity, we have chosen to present circular attributes for constant propagation that follow more closely the presentation in [1]. Further examples of circular attributes that solve data-flow optimization problems and that are based on Kildall's approach may be found in [13] and [38].

We define the domain $\mathbf{V}(v)$ of a program variable v to be the complete lattice illustrated in Figure 7, where $\bot_v$ represents the situation where v is as yet undefined in the program, the c_is represent the specific constant values that v is known to have, and $\top_v$ represents the situation where v has been assigned a value that either is not a constant or may be one of several constants. We define the domain $\mathbf{V}$ as the cross product $\mathbf{V}(v_1) \times \mathbf{V}(v_2) \times \cdots \times \mathbf{V}(v_m)$ for m variables in the program. Let $\bot$ denote the tuple $(\bot_{v_1}, \bot_{v_1}, \ldots, \bot_{v_m}) \in \mathbf{V}$, and let $\top$ denote the tuple $(\top_{v_1}, \top_{v_2}, \ldots, \top_{v_m}) \in \mathbf{V}$. Instances of each attribute in the example take values from domain $\mathbf{V}$. Given two values x and y in domain $\mathbf{V}$, we define their *least upper bound*, $x \sqcup y$, to be that least element $z \in \mathbf{V}$, such that $x \sqsubseteq z$ and $y \sqsubseteq z$. Since $\mathbf{V}(v)$ is a complete lattice for all v, so is $\mathbf{V}$, and the least upper bound is guaranteed to exist. Furthermore, the least upper bound operator is clearly monotone. Given $x \in \mathbf{V}$, a variable v_i, and an expression e, we define $gen(x, v_i, e)$ to yield a tuple $y \in \mathbf{V}$, such that the entries in y corresponding to

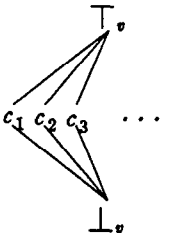

Fig. 7. Domain of a program variable.

variables v_j, for $j \neq i$, have value $\bot_{v_j}$. The entry of y corresponding to v_i has value $c \in \mathbf{V}(v_i)$, where c is the result of evaluating expression e. To evaluate the expression e, the "value" of a variable in e is taken to be its entry in x. Then the result of evaluating expression e is

$\bot_{v_i}$	if any variable v_j in the expression has value $\bot_{v_j}$,
constant c	if all variables in e have constant values and e is algebraically equal to c, and
$\top_{v_i}$	otherwise.

We note that *gen* is monotone with respect to x for fixed v_i and fixed e. The function *gen* generates a tuple in $\mathbf{V}$ that represents the effect of assigning the result of the expression to the variable v_i. Given $x \in \mathbf{V}$, and a variable v_i, we define $kill(x, v_i)$ to yield a tuple $y \in \mathbf{V}$, such that the entries in y corresponding to variables v_j, for $j \neq i$, are the same as those in x. The entry of y corresponding to v_i has value $\bot_{v_i}$. Function *kill* is monotone with respect to x for fixed v_i and effectively masks out the value v_i had before the assignment.

Nonterminal S has an inherited attribute, *in*, which represents the state of all program variables before execution of the portion of the program rooted at the derivation subtree labeled with S. S also has a synthetic attribute, *out*, which represents the state of all program variables after execution of the portion of the program rooted at the appropriate subtree.

Most of the attributes described are "copy" attributes used to distribute needed information throughout the derivation tree. We describe here only the definitions for the noncopy attributes (rules 1, 4, 8, 12). In rule 1, attribute $S'.in$ represents the state of variables before execution of the program rooted at the treenode with label S'. Since no value is assigned to the variables before program execution, each variable is considered to be undefined, and so we assign $\bot$ to $S'.in$. In rule 4, $S.in$ represents the state of all variables before execution of the body of a loop. The state of a variable v_i on entering the body of a loop can be seen to be the least upper bound of its state before entering the loop initially and its state on exit from the body. That is, v_i will be undefined only if it is undefined on initial entry and remains undefined at exit. It will be a specific constant only if it was this constant on initial entry and on exit. In any other case, it has some undetermined value. Rule 8 is similar to rule 4, but the least upper bound joins the variable states of both exits from an if-then-else two-way branch. In rule 12, the only variable whose state is affected is that represented by **id**. The function *gen* will generate a tuple representing the assignment to **id** and having value $\bot_{v_j}$

for all $v_j \neq \mathbf{id}$. The function *kill* will generate a tuple containing the current state of all variables, except that for **id**, which will be $\perp_{\mathbf{id}}$. The least upper bound of these two tuples correctly represents the states of all variables in the program after the assignment is made.

Finally, we note that attributes *S.in* and *S.out* are the only circular attributes defined and that all of the semantic functions used in defining these attributes are monotone. Furthermore, **V** is a complete lattice whose height is bounded linearly by the number of variables appearing in the program.

4. INCREMENTAL EVALUATION OF CIRCULAR ATTRIBUTE GRAMMARS

Given a consistent attributed derivation tree and a modification to the tree structure, the problem of incremental attribute evaluation is to update the attribute values and to return the tree to a consistent state. Following Reps [34], all modifications can be viewed as some combination of three basic operations: *cursor movement*, *subtree deletion*, and *subtree insertion*. The editor maintains a position in a derivation tree called the *point of modification*. Cursor movement changes the context of the edit session and corresponds to moving the point of modification up or down in the tree. Subtree deletion, shown in Figure 8, prunes the tree at the point of modification, resulting in two distinct trees: the subtree that was rooted at the point of modification, and the original tree with this subtree deleted. Subtree insertion, shown in Figure 9, splices the root of a consistent freestanding tree onto the point of modification.

We refer the reader to [34] for the details of how these operations can be carried out. Following a subtree deletion or insertion, the meaning of the attributed derivation tree may have changed, and some of the attributes at the point of modification may now be inconsistent. This change might have more than local ramifications: If we reevaluate the attributes at the point of modification in an effort to make these consistent, some of their successors in the dependency graph might become inconsistent. It is the responsibility of the incremental evaluation algorithm to return the entire attributed derivation tree to a consistent state after each modification.

4.1 Identification of Strongly Connected Components on Demand

The algorithm we present for the incremental evaluation of circular attributes takes advantage of the acyclicity of the SCC graph of $D(T)$ to establish an efficient scheduling of fixed-point computations for the circular attributes. In an attributed derivation tree containing cycles, strongly connected components may span over many treenodes. Although a modification confined to a single treenode can affect $D(T)$ only locally, it can radically alter the structure of the SCC graph, splitting or coalescing strongly connected components far beyond the affected region of the tree. For example, Figure 10 exhibits a strongly connected component that will break into many smaller strongly connected components with a single edge deletion. Such global affects make it imprudent to maintain the SCC graph. Instead, we show how it is possible to identify strongly connected components *on demand*, only as they are encountered within a region of the tree containing attributes whose values are affected by the modification.

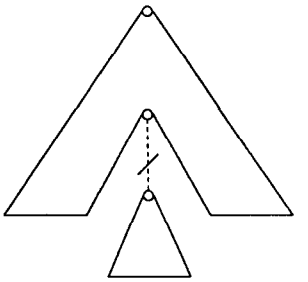

Fig. 8. Deletion of a subtree from an attributed derivation tree.

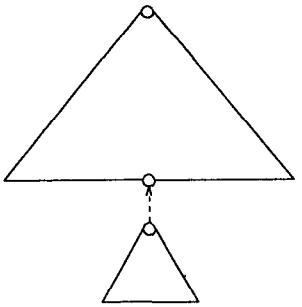

Fig. 9. Insertion of a subtree into an attributed derivation tree.

4.2 Sibling-Dependent Attributes

An attribute instance, y, occurring in a treenode, t, is said to be *sibling-dependent* if it is directly dependent on an attribute instance, x, occurring in a treenode that is a sibling of t. Sibling-dependent attributes complicate incremental evaluation by introducing dependency edges that do not conform to the structure of the tree. The sibling-dependent attributes of any attribute grammar can be removed by introducing synthetic *copy* attributes. For example, in the above definition we can introduce a copy attribute, $z = x$, in the parent treenode of t and replace every reference to x in the semantic equation for y by a reference to z (see Figure 11). To simplify the presentation and analysis of the algorithm, we assume that the attributed derivation tree contains no sibling-dependent attributes. At the end of this section, we show how this assumption can be relaxed without affecting the asymptotic behavior of the algorithm.

4.3 Definitions

An attribute instance is said to be *affected* if its final value before modification differs from its final value after modification. An attribute instance is said to be *initially influenced* if it is an attribute of the point of modification, r, and is directly or indirectly dependent on at least one attribute of the parent of r and at least one attribute of a child of r. It is readily seen that all affected attributes of r must be initially influenced. The set of initially influenced attributes provides a starting point for an incremental update. An attribute instance is said to be *influenced* if it is either initially influenced, or affected, or directly dependent on an affected attribute.

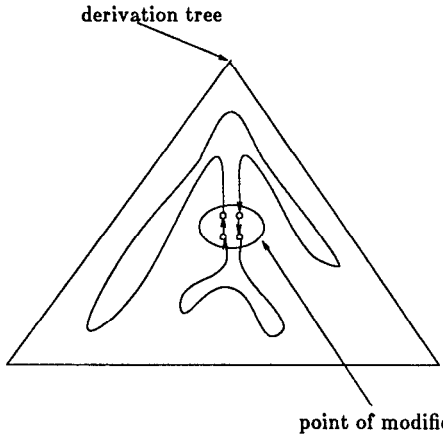

Fig. 10. A local change having global ramifications to the SCC graph.

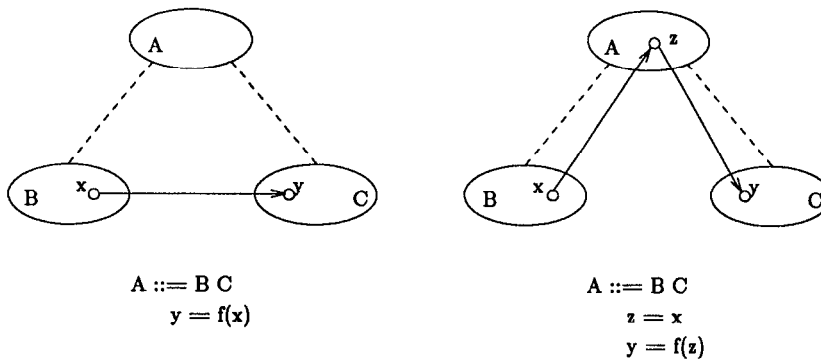

Fig. 11. Removal of a sibling-dependent attribute.

Let *AFFECTED* denote the set of all affected attributes, and let *INFLUENCED* denote the set of all influenced attributes. In general, it is not possible to determine if an attribute is affected until it has been evaluated. On the other hand, once we have evaluated an affected attribute we know that its successors are influenced and should be evaluated. The incremental evaluation algorithm for *noncircular* attribute grammars presented by Reps in [34] evaluates the attributes in *INFLUENCED*, with only O(| *INFLUENCED* |) overhead in scheduling the evaluations. Although *AFFECTED* is a subset of *INFLUENCED*, the outdegree of any derivable dependency graph is bounded by a constant for any fixed attribute grammar, and so *INFLUENCED* is O(| *AFFECTED* |) in size. Reps shows that, assuming the time for each attribute evaluation is bounded by some constant, the noncircular evaluation algorithm returns the tree to its consistent state in O(| *AFFECTED* |) time, which is optimal. In practice, attributes often have values of arbitrary size, and the corresponding semantic equations may represent computations of arbitrary complexity, violating the constant time evaluation assumption. For example, the attributes for constant propagation

presented above have values taken from a domain formed by the cross product of the domains for each variable appearing in the program. Thus, each attribute has a value whose representation may be linear in the total number of variables, which in turn may be linear in the size of the program. It follows that the cost of a single semantic function application varies from program to program and cannot be bounded. Problems arising from the use of such *aggregate* attributes and some efficient techniques for handling them are discussed further in [36]. Although in general, the constant time evaluation assumption is invalid, it serves as a good tool for simplifying the analysis of attribute evaluation strategies and for understanding the overhead involved in scheduling the evaluations.

We say a strongly connected component of $D(T)$ is affected if it contains an affected attribute, initially influenced if it contains an initially influenced attribute, and influenced if it contains an influenced attribute. Let *AFFECTEDSCC* denote the set of all affected strongly connected components, and let *INFLUENCEDSCC* denote the set of all influenced strongly connected components. The incremental evaluation algorithm that we present recomputes the fixed points of exactly the strongly connected components in *INFLUENCEDSCC* with overhead that is no more than linear in the cost of computing these fixed points.

A strongly connected component *intersects* a treenode t if it contains at least one attribute of t. It *intersects a region of tree* T if it intersects at least one treenode in the region. To determine an efficient ordering of the fixed-point computations, our algorithm considers the region of the tree containing exactly the treenodes that are intersected by influenced strongly connected components. Since this region is not known before evaluation time, it must be constructed as influenced strongly connected components are discovered.

We associate a *color* with each treenode of tree T that indicates if the treenode is intersected by a strongly connected component known to be influenced. Initially, all treenodes will be blue. As an influenced strongly connected component, c, is selected for evaluation, all treenodes intersected by c will be colored red. On termination, only treenodes intersected by influenced strongly connected components will be red, all others remaining blue. Our algorithm considers for fixed-point evaluation only strongly connected components intersecting the red region of the tree and recomputes the fixed points of only the influenced strongly connected components. In the interest of brevity, in the discussion below we say *considered* instead of *considered for evaluation.*

A strongly connected component is said to be *waiting for consideration* iff it intersects the red region of the tree and has not been considered yet. A strongly connected component is said to be *ready for consideration* iff it is waiting for consideration and is neither directly nor indirectly dependent on any other strongly connected component that is waiting for consideration. For clarity, we defer our discussion of testing for the ready-for-consideration property until a later section.

4.4 Scheduling Evaluations of Influenced Strongly Connected Components

We present in Figure 12 the algorithm for scheduling the fixed-point computations of influenced attributes. The algorithm maintains a scheduling set, S, that

```
initially all treenodes are blue and the red region is empty;
expand the red region to include r, the point of modification;
mark all initially influenced strongly connected components as influenced;
initially S is the set of all ready strongly connected components;
while S is not empty do
  select and remove a strongly connected component, b, from S;
  if b is influenced then
    expand the red region to cover all treenodes intersected by b;
    compute the fixed point of b;
    for each affected attribute, x, in b do
      for each successor, y, of x not in b do
        if y is in a blue treenode, t, then
          expand the red region to include t;
          insert into S any ready strongly connected component that intersects t;
        fi
        mark the strongly connected component containing y as influenced;
      od
    od
  fi
  insert into S any ready strongly connected component dependent on b;
od
```

Fig. 12. Scheduling fixed-point computations of influenced strongly connected components.

is exactly the set of strongly connected components that are ready for consideration. In the beginning, the initially influenced strongly connected components are all assumed to be affected and, thus, are marked as influenced, and r is colored red. We insert into S all strongly connected components intersecting r that are ready for consideration: specifically, those whose values are independent (directly or indirectly) of any other strongly connected component intersecting r.

During each iteration of the main loop, exactly one strongly connected component, say, b, is selected and considered by deleting it from S and, if marked as influenced, by computing its fixed point. Before computing the fixed point, the red region of the tree is expanded to cover all treenodes intersected by b. After the fixed-point computation, if the fixed point of b changes, then this change may have affected the fixed points of strongly connected components directly dependent on b. Not all strongly connected components directly dependent on b are influenced: only those containing attributes directly dependent on an attribute of b whose value has changed (there may be many attributes of b whose values remain unchanged). Let x be an attribute of b whose value has changed, and let y be a direct successor of x not in b. If y is in a blue treenode, t, then the red region is expanded to include t, and the strongly connected components intersecting t are checked to determine if they are ready for consideration. Since y may be affected by the change in x, the strongly connected component containing y is marked as influenced. Following consideration of b, b is no longer waiting, and we check the strongly connected components intersecting the red region that are directly or indirectly dependent on b to determine if they are now ready for consideration.

PROPOSITION 1. *The fixed point of every influenced strongly connected component will be computed.*

PROOF. Every strongly connected component that intersects the point of modification is initially waiting for consideration. Since $C(T)$ is acyclic and finite, every strongly connected component waiting for consideration will eventually become ready for consideration and, if influenced, evaluated. When evaluated, if the fixed point of a strongly connected component changes, the red region of the tree is expanded to include every direct successor that is influenced. There is a path in $D(T)$, consisting only of influenced attributes, from an initially influenced attribute to every influenced attribute. By induction on the length of this path, it can be seen that every influenced strongly connected component will eventually intersect a red treenode, become ready for consideration, and be evaluated. □

PROPOSITION 2. *Every attribute of a strongly connected component will be globally consistent following its fixed-point computation.*

PROOF. Initially, the only attributes in T that might be inconsistent are those attributes in r that are initially influenced. A strongly connected component of treenode r that is ready for consideration before the main loop is independent of any other strongly connected component intersecting r and therefore independent of any initially influenced attribute (although they may contain such attributes). Following a fixed-point computation of such strongly connected components, every attribute in the strongly connected component becomes consistent and therefore globally consistent. By induction on the length of the longest dependency path in $C(T)$ from a strongly connected component containing an initially influenced attribute, all arguments of any attribute in an influenced strongly connected component that is ready for consideration must be globally consistent. As a consequence, following any fixed-point computation, all attributes of the strongly connected component must be globally consistent. □

PROPOSITION 3. *The attributed derivation tree will be consistent when the scheduling algorithm terminates.*

PROOF. Since the fixed point of every influenced strongly connected component is computed, at termination every attribute in an influenced strongly connected component will be globally consistent. Since every attribute not in an influenced strongly connected component is globally consistent before and throughout the scheduling algorithm, when the algorithm terminates every attribute is globally consistent and the attributed derivation tree is consistent. □

4.5 Partitioning the Attributes of a Treenode

The scheduling algorithm presented above operates on an expanding region of the tree. As the red region of the tree is expanded to include a treenode t, it is necessary to partition quickly the attributes of t into their respective strongly connected components of $D(T)$. In this section we show how it is possible to partition the attributes of a single treenode into strongly connected components on demand and in constant time for any fixed attribute grammar.

Let T be an attributed derivation tree, and let $D(T)$ be its associated dependency graph. A *transitive path* is said to exist between two attributes, x and y, of a treenode, t, if and only if there is a directed acyclic path from x to y in $D(T)$ that contains at least two edges and that does not go through any attribute in t. Since there are no sibling-dependent attributes, any transitive path for a treenode t in T must exit and reenter t through attributes of a single treenode, s, that is a neighbor of t, and the path will include no attributes of any other neighbor of t. For each treenode t, define V_t to be the *transitive graph*[1] of t, such that V_t contains the edge (x, y) labeled s iff x and y are attributes of t, s is the parent or a child of t, and there is a transitive path from x to y containing at least one attribute of s. An edge (x, y) labeled s in V_t signifies that y is transitively dependent on x via a path confined through s. An edge (x, y) of V_t *goes through* treenode s if it is labeled with s. The graph V_t summarizes all paths between attributes of t that are external to t (see Figure 13).

Two attributes, x and y, in treenode t are in the same strongly connected component of $D(T)$ iff there are paths in $D(T)$ from x to y and from y to x. Let M_t be the graph defined as follows: There is an edge (x, y) in M_t iff x and y are attributes of t, and there is an edge (x, y) in either $D(T)$ or V_t. Clearly, there is a path from x to y in $D(T)$ iff there is a path from x to y in M_t. Hence, the strongly connected components of $D(T)$ that intersect t are exhibited exactly by M_t, and if we partition the attributes of t according to the strongly connected components of M_t, we have partitioned them according to the strongly connected components of $D(T)$. The graph M_t has the appealing property that its size is bounded by a constant, and therefore, finding the strongly connected components of M_t takes only constant time.

4.6 Computing and Maintaining Transitive Graphs

Because a modification to local dependencies at some treenode can affect transitive dependencies arbitrarily far away, maintaining V_t for each treenode t would require too much overhead. Instead, we define a graph $\hat{V}_t$ that is an *approximation* to V_t that contains most of the information that we require for partitioning and can be maintained with low cost. Assume that, initially, $\hat{V}_t$ is equal to V_t for all

[1] Transitive graphs are based on the *characteristic graphs* defined by Reps [34], the major differences being that

(1) each treenode has exactly one transitive graph. In [34] each treenode has both a *superior* characteristic graph summarizing transitive paths above the treenode, and a *subordinate* characteristic graph summarizing transitive paths below the treenode.
(2) transitive graphs contain no direct dependency information. In [34] the subordinate characteristic graph of a treenode includes all direct dependencies between attributes of the treenode. This information is used to construct dynamically $D(T)$ in the affected region of the tree. Our algorithm maintains $D(T)$ in its entirety, since updating $D(T)$ following a modification can be done in constant time.
(3) edges in transitive graphs are labeled, whereas edges in characteristic graphs are not. This gives more information about the transitive paths and leads to a slightly stronger *prepared for propagation* condition than that in [34]. The stronger condition, in turn, allows us to partition the attributes into strongly connected components without having to recompute transitive graphs encountered during the incremental evaluation.

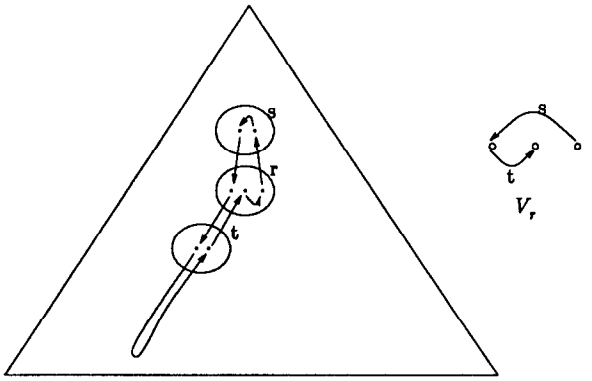

Fig. 13. A treenode and its transitive graph.

treenodes t. Let treenode s be a neighbor of t; that is, s is either the parent of t or a child of t. Following a series of modifications, the transitive graph $\hat{V}_t$ is said to be *accurate with respect to s* if there is an edge (x, y) labeled s in $\hat{V}_t$ exactly when there is a transitive path from x to y containing at least one attribute of s. An attributed derivation tree, T, is *prepared for propagation at r* if, for every treenode, t, $\hat{V}_t$ is accurate with respect to every neighbor s not on the shortest path in T between r and t.[2] This property is invariant with respect to modifications occurring at the treenode r; that is, if T is prepared for propagation at r and a modification occurs to the dependencies of attributes of r, the tree remains prepared for propagation at r. Furthermore, if a tree is prepared for propagation at r and we change the context of the edit by moving the point of modification to treenode t, which is either the parent or a child of r, $\hat{V}_t$ is the only transitive graph that might violate the prepared-for-propagation assumption, and we can reestablish the prepared-for-propagation property by recomputing only $\hat{V}_t$.

We show how $\hat{V}_t$ can be computed in constant time using only local information. $\hat{V}_t$ is accurate with respect to all neighbors of t with the possible exception of r. Define a temporary directed graph $G = (V, E)$, where V contains a vertex corresponding to each attribute in t and r, and E is the set of edges that include

(1) all edges of $\hat{V}_r$ not having label t,
(2) all edges of $D(T)$ between attributes of both t and r, and
(3) all edges of $D(T)$ between attributes of r.

$\hat{V}_t$ contains an edge (x, y) labeled s iff x and y are attributes in t, s is a neighbor of t and if s is not r, then there is an edge (x, y) labeled s in $\hat{V}_t$; otherwise, if s is r, then there is a path from x to y in G. We note that while recognizing the

[2] The primary difference between our definition of prepared for propagation and that of Reps [34] is that we maintain information about transitive paths below treenodes that are on the path between the root and the point of modification. In [34] the subordinate characteristic graphs of these treenodes are not maintained.

paths of G are equivalent to a transitive closure on G, the number of vertices and edges in G is bounded by a constant, and so we can construct V_t in constant time. By computing the transitive graph V_t and using this as $\hat{V}_t$, we reestablish the prepared-for-propagation assumption. We note that, at all times, for the point of modification r, $\hat{V}_r = V_r$.

4.7 Partitioning Revisited

We have seen that by examining the graph, M_t, the attributes of a treenode t are partitionable in constant time. The construction of M_t requires V_t, which is too expensive to maintain at each treenode t. In the previous section, we defined, for each treenode, t, a graph $\hat{V}_t$ that is an approximation to V_t and that can be maintained with only a constant overhead during a change of edit context. In this section we define a graph that is an approximation to M_t that allows us to partition the attributes of t *at the time the partitioning is required.*

Let T be an attributed derivation tree prepared for propagation at some treenode r. Let t be an arbitrary treenode in T. Consider the last time T was prepared for propagation at t. At that time, $\hat{V}_t$ was exactly V_t. Let s be that neighbor of t that the edit cursor was moved to next. Then s is on the path in T between t and all treenodes that have been modified since the last time the tree was prepared for propagation at t. Now consider the difference between V_t and $\hat{V}_t$, that is, those edges in V_t not in $\hat{V}_t$ and those edges in $\hat{V}_t$ not in V_t. Each of these edges must be labeled with s, since all transitive paths that have disappeared from or have been introduced to V_t are constrained to go through s.

The algorithm for scheduling fixed-point computations maintains the invariant that the red region of the tree is connected. Each time the red region is expanded to contain one more treenode, t, the strongly connected components intersecting t must be identified. As we pointed out in the above discussion, $\hat{V}_t$ may be inaccurate with respect to treenode s, where s is that neighbor of t on the path from t to the point of modification r. Since the red region is always connected and since r is red, at the time t is to be included in the red region, s is the only red neighbor of t and is easily identifiable. Furthermore, the attributes of s have been partitioned, and this partitioning may be used to deduce enough of the information missing from $\hat{V}_t$ to correctly partition the attributes of t.

Let $\hat{M}_t$ be the graph defined as follows: There is an edge (x, y) in $\hat{M}_t$ iff x and y are attributes of t and either

(1) (x, y) is an edge of $D(T)$;
(2) (x, y) labeled s is an edge of $\hat{V}_t$, where s is a blue treenode; or
(3) there are two attributes x' and y' (not necessarily distinct) in a red treenode, $s \neq t$, such that (x, x') and (y', y) are both edges of $D(T)$ and x' and y' are in the same strongly connected component of $D(T)$.

We show that $\hat{M}_t$ contains the information needed to partition the attributes of t according to the strongly connected components of $D(T)$ that intersect t.

PROPOSITION 4. *Two attributes of t are in the same strongly connected components of $\hat{M}_t$ iff they are in the same strongly connected components of $D(T)$.*

PROOF. The proposition can be restated as follows: For x and y in t, there are paths from x to y and from y to x in $\hat{M}_t$ iff there are paths from x to y and from y to x in $D(T)$.

($\Rightarrow$) It suffices to show that if u and v are attributes of t and (u, v) is an edge in $\hat{M}_t$, then there is a path from u to v in $D(T)$. This is clearly true by the definition of $\hat{M}_t$.

($\Leftarrow$) We may break a path from x to y (or y to x) into segments of two distinct types. In a type one segment, every edge of the segment is between attributes of t. In a type two segment, the first edge of the segment is from an attribute of t to an attribute not in t, the last edge is from an attribute not in t to an attribute in t, and all other edges of the segment are between attributes not in t. Clearly, if there is a type one segment from u to v then there must be a path from u to v in $\hat{M}_t$. Let there be a type two segment from u to v in $D(T)$. This segment must go through some treenode s that is either the parent or child of t. If s is blue, then $\hat{V}_t$ is accurate with respect to s, that is, there is an edge (u, v) in $\hat{V}_t$ labeled s, and by definition, (u, v) is also in $\hat{M}_t$. If s is red, then there are attributes u' and v' in s, such that there are edges (u, u'), (v', v) in $D(T)$ and a path from u' to v'. Since all attributes on any path from x to y or from y to x are in the same strongly connected component, u' and v' are in the same strongly connected component, and there is an edge (u, v) in $\hat{M}_t$. By connecting the paths in $\hat{M}_t$ corresponding to each segment, we may construct a path from x to y and one from y to x in $\hat{M}_t$. □

Initially, all treenodes are blue, so that $\hat{M}_r$, for the point of modification r, can be constructed without knowledge of the strongly connected components of any neighbors of r in the tree. This is because the tree is prepared for propagation at r, and so $\hat{M}_r$ is exactly M_r. In general, a treenode will be colored red only after its attributes have been partitioned according to their strongly connected components. At the time $\hat{M}_t$ is required for some treenode t with a red neighbor, s, the attributes of s will have been partitioned into their corresponding strongly connected components.

Given a treenode, t, we can partition the attributes of t using depth-first search in time proportional to the sum of the number of vertices and edges of $\hat{M}_t$, which is bounded by a constant. We note that $\hat{M}_t$ does not need to be explicitly constructed to do the partitioning. Instead, we can arrange for the depth-first search to follow the edges of $D(T)$ and $\hat{V}_t$ according to the definition of $\hat{M}_t$. For each attribute x of treenode t, let C_x denote the *partition containing* x, that is, the set of all attributes of t that are in the same strongly connected component of $D(T)$ as x. Let P_t denote the *partitioning* of t; that is, P_t is the set of all distinct partitions C_x such that x is an attribute of treenode t.

4.8 Modeling Dependencies

Partitioning the attributes of a region of the attributed derivation tree identifies the attributes that should be treated as a unit during a fixed-point computation, but partitioning by itself gives no clue to the order in which these computations

should be performed. In exhaustive evaluation, the dependencies between the strongly connected components are reflected in $C(T)$, and a topological ordering of the vertices in $C(T)$ yields an efficient ordering of the fixed-point computations. In incremental evaluation it is too expensive to derive $C(T)$. Instead, transitive graphs are used to supplement the direct dependencies of $D(T)$ in the influenced region with summary information about the dependencies occurring outside the region, and a topological traversal of only the strongly connected components intersecting this region is effected.

We construct a directed graph, M, that models the direct and transitive dependencies of the red region of the tree. Each strongly connected component intersecting the red region is represented by a single vertex in M. Dependencies are represented by *direct* and *transitive* edges in M. A direct edge (b, c) is in M iff b and c are two distinct strongly connected components of $D(T)$ and there are attributes x and y in b and c, respectively, such that (x, y) is in $D(T)$ and both x and y are in red treenodes. There is a transitive edge (b, c) labeled s in M iff b and c are two distinct strongly connected components of $D(T)$ and there are attributes x and y in b and c, respectively, such that (x, y) labeled s is in $\hat{V}_t$ for some red treenode t and blue treenode s. A direct or transitive edge (b, c) represents the functional dependence of c's fixed point on b's fixed point.

M is constructed to model the dependencies in an expanding region of the tree. Consider M when the red region contains only r, the point of modification. Each partition of r corresponds to a distinct strongly connected component of $D(T)$ and is represented by a vertex in M. There is a direct edge (b, c) in M iff there are attributes x and y in distinct partitions of r, b and c represent the strongly connected components corresponding to C_x and C_y, respectively, and there is an edge (x, y) in $D(T)$. There is a transitive edge (b, c) labeled s in M iff there are attributes x and y in distinct partitions of r, b and c represent the strongly connected components corresponding to C_x and C_y, respectively, and there is a transitive edge (x, y) labeled s in $\hat{M}_r$.

The problem is more complicated when M is nonempty and we expand the red region to include a new treenode t. Some of the partitions in t may correspond to strongly connected components that are already represented by a vertex in M. Such situations can be identified by examining the edges of $D(T)$ between the attributes in the partitions of t and in the partitions of the red treenode adjacent to t. If these edges form a cycle involving exactly two such partitions, then the attributes of both are in the same strongly connected component and should be represented in M by a single vertex. To test a single partition $C \in P_t$, we examine $D(T)$ and determine the set of partitions of s that contain attributes that are direct successors of the attributes in C. Next, $D(T)$ is examined for any edges leading from an attribute in a partition of s discovered above to an attribute of C. If such an edge is found, then there is already a vertex in M that represents the strongly connected component corresponding to C. Otherwise, we introduce a new vertex in M to represent this strongly connected component.

The algorithm presented in Figure 14 expands M to include dependencies involving the strongly connected components that intersect treenode t. First, the attributes of t are partitioned into their strongly connected components. Second, each partition of t is represented by a vertex in M. If the partition corresponds

```
Expand(t):
  let s be t's red neighbor in the tree;
  partition the attributes of t using depth-first search on M̂_t;
  for each partition C ∈ P_t do
    if the strongly connected component corresponding to C is not in M then
      introduce a new vertex in M to represent this strongly connected component;
    fi
  od
  for each edge (x,y) in D(T) between attributes in t or between those in t and s do
    if x and y are in different strongly connected components then
      insert a direct edge in M from the vertex representing the strongly connected
      component containing x to the vertex representing the strongly connected com-
      ponent containing y;
    fi
  od
  for each edge (x,y) labeled u in V̂_t such that u is a blue treenode do
    if x and y are in different strongly connected components then
      insert a transitive edge labeled u in M from the vertex representing the strongly
      connected component containing x to the vertex representing the strongly con-
      nected component containing y;
    fi
  od
  color t red;
  delete all transitive edges in M with label t;
end Expand;
```

Fig. 14. Expanding the red region to include treenode t.

to a strongly connected component already represented in M, then it is identified with the appropriate vertex. Otherwise, a new vertex is inserted into M to represent the corresponding strongly connected component, and the partition is identified with this new vertex. Third, the dependencies between the strongly connected components in the new region are extracted. Dependencies between attributes in distinct strongly connected components of $D(T)$ are reflected as dependencies between the corresponding vertices of M. Direct edges are found by examining edges of $D(T)$ occurring in the region of t. Transitive edges are found by examining only the blue edges of $\hat{V}_t$. Finally, treenode t is colored red, and any transitive edges in M that go through t are deleted.

4.9 Covering a Strongly Connected Component with M

Depending on the scope of the modification made, the semantic equations, or the values of the attributes involved, an evaluation algorithm used for updating the fixed point of a strongly connected component may require evaluating every member attribute. In anticipation of such cases and to simplify whatever evaluation algorithm is applied, whenever it is determined that the fixed point of a strongly connected component must be computed, M is expanded to cover all treenodes intersected by the strongly connected component.

The algorithm presented in Figure 15 selects the treenodes requiring expansion before the fixed point of the strongly connected component corresponding to b is computed. Let X be the set of all treenodes covered by M and intersected by b. This set may be determined by examining the set of attributes associated with b.

```
Cover(b):
  let X be the set of all red treenodes intersected by b;
  while X is not empty do
    select and remove a treenode t from X;
    for each edge (x,y) labeled s in V̂_t such that s is a blue treenode do
      if x and y are in b then
        Expand(s);
        insert s into X;
      fi
    od
  od
end Cover;
```

Fig. 15. Expanding M to cover a strongly connected component b.

Let t be a treenode in X, and let s be a blue treenode adjacent to t that also contains an attribute of b. By the connectivity of strongly connected components, there exist attributes x and y in t such that there is a path from x to y through s. Such a path occurs iff there is an edge (x, y) labeled s in $\hat{V}_t$. If this is the case, s is expanded and added to the set X.

The covering algorithm takes time proportional to the number of treenodes intersecting b, having an optimal worst-case time complexity of $O(k_b)$, where k_b is the number of attributes in b. It should be pointed out that the number of influenced attributes of the influenced strongly connected component b may be smaller than the number of attributes in b. Expanding all treenodes that intersect an influenced strongly connected component may not be an optimal strategy: Under some circumstances, it may be possible to update the fixed point of a strongly connected component without visiting every attribute. Under these conditions, a dynamic strategy, such as that described in [21], where treenodes are covered by M only after they are found to contain influenced attributes, may be faster.

4.10 Testing for the Ready Condition

The ready-for-consideration property for a strongly connected component c can now be easily checked by examining the edges with head c in M. Recall that c is ready for consideration iff it is waiting for consideration (i.e., if it is represented in M) and it is neither directly nor indirectly dependent on any other strongly connected component that is waiting for consideration. Suppose that c is waiting for consideration and directly dependent on some strongly connected component, b, that is waiting for consideration. Then there is an edge (x, y) in $D(T)$ such that x is an attribute of b and y is an attribute of c. If x and y are both in red treenodes, then there will be a direct edge (b, c) in M. If either x or y or both are in blue treenodes, then there are attributes x' in b and y' in c, such that both are in red treenodes, and there is a path from x' to y' *that lies wholly outside the red region of the tree* and that contains the edge (x, y). Then there will be a transitive edge (b, c) labeled s in M for some blue treenode, s. Now suppose that c is not ready for consideration and is not directly dependent on any strongly connected component that is waiting for consideration. Then c must be indirectly dependent on some strongly connected component, b, waiting for consideration,

and there are attributes x in b and y in c such that there is a path from x to y in $D(T)$ that lies wholly outside the red region of the tree. Again, there will be a transitive edge (b, c) labeled s in M for some blue treenode s.

There are two distinct times when it is necessary to check for strongly connected components that have just become ready for consideration. After a treenode changes from blue to red, every strongly connected component intersecting this treenode is checked with a call to Ready. After a strongly connected component, b, is considered, there may be some strongly connected components that are either directly or indirectly dependent on b that are now ready for consideration. Since these must intersect a red treenode and be independent of other strongly connected components that are waiting for consideration, a strongly connected component, c, need only be checked if there is either a direct or transitive edge (b, c) in M.

The algorithm presented in Figure 16 tests a strongly connected component, c, for the ready-for-consideration property by examining the edges in M. The procedure Ready returns true if c is ready for consideration, and false otherwise. Ready returns as soon as it has encountered an edge that indicates c is not ready. Edges in M that no longer keep c from becoming ready are deleted as they are encountered. The cost of each call to Ready associated with a given strongly connected component, c, is proportional to the number of edges in M that are deleted during that call. Since an edge in M, once deleted, is never reinserted, the total cost of all calls associated with c is proportional to the indegree of c in M.

4.11 Analysis of the Incremental Evaluation of Circular Attribute Grammars

We may now analyze the algorithm for the incremental evaluation of circular attribute grammars. Let r be the point of modification. Before the beginning of execution, the graph M is empty, and all treenodes are blue.

At the beginning of the algorithm, the red region of the tree is expanded to include r; that is, the attributes of r are partitioned via a depth-first search on the graph $\hat{M}_r$, and the appropriate vertices and edges are introduced into M. All initially influenced attributes are then identified by examining the edges of $D(T)$ involving attributes of r, and the strongly connected components containing these attributes are marked as influenced. As a last step before the main loop, the set of strongly connected components of r is examined to determine those components that are ready for consideration. For a fixed attribute grammar, the indegree of $D(T)$ is bounded by a constant, as is the number of attributes in any treenode of T. Then $\hat{M}_r$ is constructible and partitionable in constant time, and the number of strongly connected components intersecting r is bounded by a constant. Identifying initially influenced attributes takes time linear in the number of edges involving attributes of r, which is bounded by a constant. At this point, the size of the graph M is bounded by a constant, each call to Ready takes constant time, and there are only a constant number of calls. It follows that all operations before the main loop take time O(1).

We now consider the cost of each iteration of the main loop. For each iteration, a strongly connected component is selected for consideration, removed from the set S, and never reinserted. Thus, each strongly connected component will be

```
Ready(c):
  for each direct edge (b,c) in M do
    if b is waiting for consideration then
      return false;
    fi
    delete direct edge (b,c) from M;
  od
  for each transitive edge (b,c) labeled s in M do
    if b is waiting for consideration then
      return false;
    fi
    delete transitive edge (b,c) labeled s from M;
  od
  return true;
end Ready;
```

Fig. 16. Testing strongly connected component c for the ready property.

considered exactly once. To simplify the analysis, we charge each iteration with the cost of all work that was required to determine when the strongly connected component was ready for consideration, removing this cost from the iterations that did the work.

Let b be the strongly connected component selected for consideration. If b is influenced, then the red region of the tree is expanded to cover all treenodes intersected by b, and b's fixed point is computed. In the previous section, we saw that the cost of this expansion is proportional to the number of attributes in b. Following the fixed-point computation, each attribute, x, in b is examined. If x is affected, the red region is expanded to include all blue treenodes containing successors of x not in b. All strongly connected components containing such successors of x are influenced and are so marked. For a fixed attribute grammar, the outdegree of $D(T)$ is bounded by a constant, and so the work following the fixed-point computation is proportional to the number of attributes in b. Finally, we must charge the iteration with the cost of determining when b was ready for consideration, which is proportional to the indegree of b in M.

Each direct edge in M with head b corresponds to a direct edge in $D(T)$ with head y, such that y is an attribute of b. Each transitive edge in M with head b corresponds to a transitive edge of $\hat{V}_t$ for some red treenode t intersected by b. Since both the indegree of $D(T)$ and that of $\hat{V}_t$ for any t are bounded by a constant, the indegree of b in M is at most linear in the number of attributes in b. If we assume that the cost of recomputing the fixed point of b is at least linear in the size of b, then the total cost of considering and recomputing the fixed point of an influenced strongly connected component is dominated by the cost of the fixed-point computation. Let $|F_i|$ denote the cost of computing the fixed point of the ith member of *INFLUENCEDSCC*, and let m be the number of influenced strongly connected components. Then the cost of considering and recomputing the fixed points of all influenced strongly connected components is $O(\sum_{i=1}^{m} |F_i|)$.

If a strongly connected component, b, is not influenced, then the total cost of considering b is the charge for determining when b was ready for consideration, which is proportional to the indegree of b in M. Then the total cost for considering all strongly connected components that are not influenced is at most linear in the number of edges in M. The indegrees of $D(T)$ and $\hat{V}_t$ are bounded by

constants, and so the number of edges in M is at most linear in the number of attributes of all influenced strongly connected components. Again, assuming that each fixed-point computation is at least linear in the size of the strongly connected component, the cost of considering all strongly connected components that are not influenced is $O(\sum_{i=1}^{m} |F_i|)$.

We have shown that the algorithm we have presented for the incremental evaluation of circular attribute grammars has a worst-case running time bounded by $O(\sum_{i=1}^{m} |F_i|)$, where $|F_i|$ is the cost of recomputing the fixed point of the ith influenced strongly connected component. This analysis makes the assumption that recomputing a fixed point takes time at least linear in the size of the strongly connected component. Under some conditions, it may be possible to update the fixed point of a strongly connected component without visiting every member attribute. A more general analysis yields a worst-case behavior of $O(\sum_{i=1}^{m} (k_i + |F_i|))$, for k_i, which is the number of attributes in the ith influenced strongly connected component. We note that, in the special case where either the attribute grammar is noncircular or the specific dependency graph under consideration is acyclic, $m = O(|AFFECTED|)$, $k_i = 1$, for all i, and the algorithm reduces to the standard one of [34], which has an optimal worst-case behavior $O(|AFFECTED|)$, under the assumption that the cost of evaluating any attribute is bounded by a constant.

4.12 A Final Note on Sibling-Dependent Attributes

The incremental evaluation algorithm presented assumes that the attributed derivation tree contains no sibling-dependent attributes. This constraint can be relaxed without changing the asymptotic behavior of the algorithm, although at the cost of some overhead. Two key observations are necessary. First, dependencies between attributes of sibling treenodes may introduce transitive dependencies between the attributes of a treenode and its parent, causing inaccuracies in the computation of $\hat{V}_t$. Second, applying the above algorithm to attributed derivation trees containing sibling-dependent attributes may invalidate the assumption that the red region of the tree is connected. The first problem can be solved by changing the computation of $\hat{V}_t$ to include the dependencies of t's siblings. The second problem can be solved by maintaining the invariant that, if one child of a red treenode, t, is red and that child contains a sibling-dependent attribute, then all children of t are red, thus guaranteeing the connectivity of the red region. By incorporating the above changes into the algorithm, the red region of the tree remains proportional to the number of treenodes intersected by influenced strongly connected components; however, some of these treenodes may not be intersected by influenced strongly connected components. We note that the incremental algorithm of [34] takes a similar philosophy: If both a treenode t and one of its children are in the region under consideration, all children of t are in the region under consideration. It follows that the algorithm of [34] considers for evaluation attributes of treenodes containing no influenced attributes. By removal of sibling-dependent attributes, it is possible to obtain an incremental algorithm for noncircular attribute grammars that considers only attributes in influenced treenodes [20]. We point out that, for the incremental

evaluation of both circular and noncircular attribute grammars, the introduction of copy attributes to remove sibling-dependent attributes may often be more desirable than using a more general algorithm that handles sibling-dependent attributes, but that operates on a larger region of the tree.

5. CONCLUSIONS

Attribute grammars have proved to be an elegant tool for coupling the static semantics of a programming language to its syntax, as an aid in the automatic generation of programming language systems. The original definition proposed by Knuth prohibited the use of circular attribute grammars in an effort to guarantee that the attribute grammar be well defined, that is, that there always be a consistent assignment to the attribute instances, regardless of the particular derivation tree under consideration. With only few exceptions, the use of attribute grammars appearing in the literature has been restricted to the noncircular ones. While noncircularity is a sufficient condition for guaranteeing a consistent assignment, it is not a necessary condition. It suffices that all circularly defined attributes have a fixed point that can be achieved with finite computation. The use of circularity in attribute grammars can improve the expressiveness of problems requiring iterative solutions. Examples of problems that are naturally solved using circular attribute grammar techniques include data-flow analysis, run-time semantics, and circuit simulation.

In this paper we have viewed the evaluation of circular attribute grammars as a fixed-point computation. Naive evaluation methods applied to attribute dependency graphs containing cycles may result in spurious evaluations and slow convergence. We have shown how the size of the fixed-point computations can be limited by partitioning the attribute instances into interdependent sets. The partitioning process yields a directed acyclic graph expressing the dependencies between the partitions, which can then be used to obtain an efficient ordering of the fixed-point computations. Applying the above strategies to the standard evaluation algorithms for noncircular attribute grammars yields efficient evaluation algorithms for circular attribute grammars. The resulting batch and incremental algorithms have worst-case running time that is proportional in the cost of computing the fixed points of only those partitions containing affected attributes or attributes directly dependent on affected attributes. When the attribute grammar is noncircular, or the specific dependency graph under consideration is acyclic, both algorithms reduce to the standard optimal algorithms for noncircular attribute evaluation. This suggests that the methods presented in this paper are not only relevant to circular attribute grammars, but also to noncircular attribute grammars whose noncircularity has not been established.

The algorithms for circular attribute grammars presented are examples of dynamic evaluation methods, since they determine an efficient ordering among the attribute instances during the evaluation process. Experience with noncircular attribute grammars has shown that static evaluation methods (e.g., those of Kastens [23] and Yeh [44]) that establish an efficient evaluation ordering during system generation time usually outperform dynamic methods. Although circular attribute grammars, by definition, cannot be completely ordered, some

ordering of the attributes may be possible. It is likely that static evaluation algorithms for circular attribute grammars derived from those for noncircular attribute grammars will exhibit superior run-time performance. Another approach that might be useful for deriving fast methods for circular attribute grammars is to use an approximate topological ordering scheme, as presented by Hoover [16], which may evaluate more attributes than necessary, but usually outperforms other methods.

ACKNOWLEDGMENTS

I am indebted to Janos Simon, whose advice was critical throughout much of this research. This paper has also benefited from a number of conversations with Thomas Reps. Finally, I would like to thank the editor and reviewers for their many helpful comments and suggestions.

REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. ALBLAS, H. Incremental simple multi-pass attribute evaluation. In *Proceedings of the NGI/SION 1986 Symposium.* 1986, pp. 319–342.
3. BABICH, W. A. High level data flow analysis using a parse tree representation of the program. Ph.D. thesis, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, 1977.
4. BABICH, W. A., AND JAZAYERI, M. The method of attributes for data flow analysis. Part I: Exhaustive analysis. *Acta Inf. 10*, 3 (Oct. 1978), 245–264.
5. BAHLKE, R., AND SNELTING, G. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst. 8*, 4 (Oct. 1986), 547–576.
6. BESHERS, G. M., AND CAMPBELL, R. H. Maintained and constructor attributes. In *Proceedings of the 1985 Symposium on Language Issues in Programming Environments. SIGPLAN Not. (ACM)* 20, 7 (June 1985), 34–42.
7. BOCHMANN, G. V. Semantic evaluation from left to right. *Commun. ACM 19*, 2 (Feb. 1976), 55–62.
8. CHEBOTAR, K. S. Some modifications of Knuth's algorithm for verifying cyclicity of attribute grammars. *Program. Comput. Softw. 7*, 1 (Feb. 1981), 58–61.
9. DEMERS, A., REPS, T., AND TEITELBAUM, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 1981). ACM, New York, 1981, 105–116.
10. DEMERS, A., ROGERS, A., AND ZADECK, F. K. Attribute propagation by message passing. In *Proceedings of the 1985 Symposium on Language Issues in Programming Environments. SIGPLAN Not. (ACM)* 20, 7 (June 1985), 43–59.
11. DERANSART, P., JOURDAN, M., AND LORHO, B. Speeding up circularity tests for attribute grammars. *Acta Inf. 21*, 4 (Nov. 1984), 375–391.
12. FARROW, R. W. LINGUIST-86: Yet another translator writing system based on attribute grammars. In *Proceedings of the 1982 Symposium on Compiler Construction. SIGPLAN Not. (ACM)* 18, 6 (June 1983), 160–171.
13. FARROW, R. W. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the 1986 Symposium on Compiler Construction. SIGPLAN Not. (ACM) 21*, 7 (July 1986), 85–98.
14. FISCHER, C. N., JOHNSON, G. F., MAUNEY, J., PAL, A., AND STOCK, D. L. The Poe language-based editor project. In *Proceedings of the 1984 Software Engineering Symposium on Practical Software Development Environments. SIGPLAN Not. (ACM) 19*, 5 (Apr. 1984), 21–29.

15. GANZINGER, H., GIEGERICH, R., MÖNCKE, U., AND WILHELM, R. A truly generative semantics-directed compiler generator. In *Proceedings of the 1982 Symposium on Compiler Construction. SIGPLAN Not. (ACM) 17*, 6 (June 1982), 172–184.
16. HOOVER, R. Dynamically bypassing copy rule chains in attribute grammars. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan.). ACM, New York, 1986, 14–25.
17. JAZAYERI, M., OGDEN, W. F., AND ROUNDS, W. C. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Commun. ACM 18*, 12 (Dec. 1975), 697–706.
18. JOHNSON, G. F., AND FISCHER, C. N. Non-syntactic attribute flow in language based editors. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 1982). ACM, New York, 1982, 185–195.
19. JOHNSON, G. F., AND FISCHER, C. N. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 1985). ACM, New York, 1985, 141–151.
20. JONES, L. G. Incremental VLSI design systems based on circular attribute grammars. Ph.D. thesis, Computer Science Dept., Pennsylvania State Univ., University Park, Pa., 1986.
21. JONES, L. G., AND SIMON, J. Hierarchical VLSI design systems based on attribute grammars. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 1986). ACM, New York, 1986, 58–69.
22. JOURDAN, M., AND PARIGOT, D. More on speeding up circularity tests for attribute grammars. INRIA Rapports de Recherche No. 828, 1988 Institut National de Rocherche en Inform, Rocquencourt, France.
23. KASTENS, U. Ordered attribute grammars. *Acta Inf. 13*, 3 (March 1980), 229–256.
24. KASTENS, U., HUTT, B., AND ZIMMERMAN, E. *GAG: A Practical Compiler Generator. Lecture Notes in Computer Science, vol. 141.* Springer-Verlag, New York, 1982.
25. KENNEDY, K., AND WARREN, S. K. Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of the 3rd ACM Symposium on Principles of Programming Languages* (Atlanta, Ga., Jan. 1976). ACM, New York, 1976, 32–49.
26. KILDALL, G. A. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Boston, Mass., Oct. 1973). ACM, New York, 1973, 194–206.
27. KNUTH, D. E. *The Art of Computer Programming. Vol. 1., Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
28. KNUTH, D. E. Semantics of context-free languages. *Math. Syst. Theory 2*, 2 (June 1968), 127–145.
29. KNUTH, D. E. Semantics of context-free languages: Correction. *Math. Syst. Theory 5*, 1 (Mar. 1971), 95–96.
30. LEWIS, P. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. Attributed translations. *J. Comput. Syst. Sci. 9*, 3 (Dec. 1974), 279–307.
31. LORHO, B., AND PAIR, C. Algorithms for checking consistency of attribute grammars. In *Proving and Improving Programs. Symposium IRIA.* (Arc-et-Senans, France) 1975, 29–54.
32. RÄIHÄ, K. J., AND SAARINEN, M. Testing attribute grammars for circularity. *Acta Inf. 17*, 2 (June 1982), 185–192.
33. REPS, T. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages* (Albuqurque, N.M., Jan. 1982). ACM, New York, 1982, 169–176.
34. REPS, T. *Generating Language-Based Environments.* MIT Press, Cambridge, Mass., 1984.
35. REPS, T., AND TEITELBAUM, T. The synthesizer generator. In *ACM Software Engineering Symposium on Practical Software Development Environments* (Apr. 1984). ACM, New York, 1984, 42–48.
36. REPS, T., MARCEAU, C., AND TEITELBAUM, T. Remote attribute updating for language-based editors. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (Jan. 1986). ACM, New York, 1986, 1–13.
37. 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.

38. SKEDZELESKI, S. K. Definition and use of attribute reevaluation in attributed grammars. Tech. Rep. 340, Dept. of Computer Science, Univ. of Wisconsin, Madison, Wisconsin, 1978.
39. STOY, J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass., 1977.
40. TARJAN, R. E. Depth first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (June 1972), 146–160.
41. TEITELBAUM, T., AND REPS, T. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sept. 1981), 563–573.
42. WAITE, W. M., AND GOOS, G. *Compiler Construction.* Springer-Verlag, New York, 1984.
43. WALZ, J. A., AND JOHNSON, G. F. Incremental evaluation for a general class of circular attribute grammars. In *Proceedings of the 1988 Conference on Programming Language Design and Implementation* (Atlanta, Ga., June 1988). *SIGPLAN Notices, 23*, 7, 1988, 308–320.
44. YEH, D. On incremental evaluation of ordered attributed grammars. *BIT 23*, 3 (1983), 308–320.

Received May 1988; revised February 1989; accepted November 1989