Incremental Evaluation for a General Class of Circular Attribute Grammars

Janet A. Walz
Cornell University (*in absentia*)
Ithaca, NY 14853

Gregory F. Johnson
University of Maryland
College Park, MD 20742

1. Introduction.

The strengths of attribute grammars as a basis for programming environments are well known. Several environment research projects rely on attribute grammars to support incremental semantic analysis ([ReT84], [FJM83], [BaS86]). Attribute grammars provide a high-level, declarative style for describing the static semantics of programming languages. Further, such descriptions are amenable to automated analysis and the production of programming environments that incrementally perform various semantic analyses of programs. Such automatically generated environments have several desirable properties; even though the writer of an attribute grammar need not be concerned with editor-time, dynamic issues such as order of attribute evaluation, an editor generator can produce environments that incrementally re-evaluate attribute values in an optimal fashion after user changes to the program ([RTD83]).

Attribute grammars have traditionally been required to be noncircular; that is, no parse tree derivable from the associated context-free grammar is allowed to give rise to cyclical or circular functional dependencies among attributes. In fact, the main result of the seminal paper on attribute grammars, [Knu68], is an algorithm for testing attribute grammars for noncircularity. However, several researchers have discovered applications in which relaxation of the requirement of non-circularity has given rise to natural, elegant solutions. Among these applications are instruction selection for code generation ([Ske78]), control and data flow analyses ([Far86]), and VLSI design problems ([JoS86]). Attention has recently been given to incremental attribute evaluation in the presence of circular functional dependencies in order to make it possible for programming environments to be based on such circular attribute grammars. Results obtained previously have imposed various restrictions such as monotonicity of evaluation functions and domains that are lattices of finite height ([JoS86]) in order to assure termination of the evaluation process. However, there are some important classes of functions for which termination is not assured, such as interpreters. The work described herein is motivated by a desire to obtain optimal or near optimal incremental re-evaluation behavior for this important category of problems.

If a user is manipulating a fairly large or computationally expensive program and asks for it to be executed by an interpreter in the environment, then subsequently modifies it and asks for an interpretation of the slightly modified program, the program should be re-interpreted in a minimal way, preserving as much information as possible from the previous execution, rather than

This work was partially supported by the Air Force Office of Scientific Research.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

being re-interpreted from the beginning. This approach contrasts, for instance, with the technique of using a noncircular attribute grammar to produce code that is then interpreted or directly executed; in that technique, code is incrementally kept in agreement with the user's program, but interpretation or execution always starts from the beginning rather than making use of the results of previous interpretations. In another related approach [BMS87], interpreters and debuggers are automatically generated from denotational descriptions of programming language semantics, but subsequent re-interpretations of a possibly slightly modified programs again result in re-execution from the beginning. The new approach described here requires use of stores as attributes; since there may be several distinct store-valued attributes in a parse tree, a form of version management aids efficiency.

Our approach provides a general, rigorous, generator-based framework which addresses many of the same issues as the hand-coded system of [KaW87]. This approach gives rise to a very appealing VisiCalc-like style of programming, in which every program modification causes fast update of the contents of input/output windows that show the result of program execution on sample inputs. Our implementation was built on the POE system [FJM83], an attribute grammar based programming environment. The addition of run-time semantics via circular attribute grammars permits automatically generated environments to be fairly complete, in that incremental static semantic checking and fast incremental execution are now available within a single framework.

The approach we employ is based on a new category of attribute grammars called *gated attribute grammars*. In a gated attribute grammar, every cycle in an attribute dependency graph must have a *gate* attribute that specifies the location in the cycle at which evaluation is to start. To evaluate the attributes in a tree, one temporarily views strongly connected components as single nodes and evaluates the nodes of the collapsed graph in topological order. Evaluation of a strongly connected component involves an inductive application of this scheme: one removes incoming arcs to the gate attribute of the outer SCC to obtain an acyclic graph, identifies nested strongly connected components, and considers these SCCs to be individual nodes. The evaluation of this graph then proceeds in accordance with a topological ordering on its nodes.

The distinguishing characteristic of a gate attribute is that it has two evaluation functions. The first evaluation function has as inputs only attributes outside the gate attribute's SCC and is used to provide an initial value to the gate. The second evaluation function depends on attributes in the SCC, and is used to obtain the succession of subsequent gate values. A boolean valued pseudo-attribute named *start* is associated with a gate attribute to determine which of the gate's evaluation functions should be invoked when it needs to be evaluated. Evaluation of the SCC is complete when (and if) it reaches a fixed point; that is, when every attribute in the SCC contains a value that is the same as the result produced by its evaluation function when it is applied to the values of the attribute's predecessors.

As with [JoS86], incremental evaluation at the outermost level can be accomplished using results from [RTD83] and subsequent refinements. A major difference, however, arises when an input to a strongly connected component "node" changes value. We do not merely want to schedule the attribute that is the successor of the changed attribute for re-evaluation, since that node might be in the middle of the SCC. Rather, we also schedule the gate of the SCC for re-evaluation,

since that is the attribute at which evaluation of the SCC is supposed to start. At evaluator time we employ nonlocal productions ([JoF85], [Hoo86], [RMT86]) to attach inputs of an SCC to the start attribute of the SCC. The purpose of these connections is to assert that the start attribute of the SCC is an evaluation-time successor of each input to the SCC. The writer of an attribute grammar designates certain attributes as gates and gives them two evaluation functions. The whole technique results, we believe, in an extended attribute grammar notation applicable to a large and important class of problems that are most naturally expressed using circular attribute grammars, and from which efficient programming environments can be automatically created.

In the following sections we define the class of gated attribute grammars and the notion of gated strongly connected components; this latter concept is critical to the approach presented herein. We describe the initial and incremental evaluation algorithms for gated attribute grammars; the approach is, we believe, quite general, and would be readily implementable in any of a variety of attribute grammar based editor and compiler generating tools.

2. Circular Attribute Grammars.

An attribute grammar is a context-free grammar in which each symbol may have associated attributes and each production may have associated functions which define certain attributes of symbols in the production in terms of others. An attributed parse tree is a parse tree of the CFG in which each instance of a grammar symbol has instances of the attributes associated with that symbol, and the attribute instances are connected into a graph by the functional dependency relations imposed by the production instances.

Traditionally, work on attribute grammars has focused on those grammars for which every possible attribute dependency graph is acyclic. Under this condition, a single traversal, in topological order, of the dependency graph can consistently attribute the entire tree. Each attribute's defining function will be evaluated exactly once in this traversal, guaranteeing that the attribution process will terminate, as long as each defining function itself terminates.

Global topological information for ordering attribute evaluation can be delivered to each instance of a grammar symbol by superior and inferior characteristic graphs. An inferior characteristic graph shows the projection of transitive dependencies from the portion of the parse tree below the symbol; a superior characteristic graph shows the projection of transitive dependencies from the remainder of the tree. These characteristic graphs, combined with the local dependencies imposed by production instances, allow local evaluation to proceed in accordance with global topological order.

Such characteristic graphs also allow optimal incremental updating after some portion of the parse tree has been changed. When superior characteristic graphs are kept for symbol instances between the cursor position, which is the point of change, and the root of the parse tree, and inferior characteristic graphs are kept for all other instances, [RTD83] show how updating after a subtree replacement can be accomplished in O(|AFFECTED|), where AFFECTED is defined as the set of attributes whose values after updating quiesces differ from their values before the editing change. If an attribute grammar is partitionable (the class of ordered attribute grammars being a polynomially-recognizable subset of the partitionable attribute grammars [Kas80]), then the above optimality results are obtainable without the

expense of maintaining characteristic graphs at evaluation time.

More recently, several people have done work relaxing the total ban on cycles in the attribute dependency graph. [Ske78] was one of the first researchers to explore this area. [JoS86] allow arbitrary dependency graphs, but restrict the defining functions for attributes that may appear in strongly connected components of a dependency graph. These functions are required to be monotonic and to take values from a lattice of finite height. By treating each maximal strongly connected component as an attribute in a collapsed graph for SCC scheduling, they maintain a similar worst-case incremental update time, $O(hk|AFFECTEDSCC|)$, where h is the height of the highest lattice and k is the largest number of nodes in an affected SCC. They also show that the local portion of the collapsed dependency graph can be constructed from the locally available characteristic graphs and production dependencies.

[Far86] also allows arbitrary dependency graphs and restricts the defining functions in strongly connected components to be monotonic, but instead of requiring these functions to operate on finite-height lattices, he requires them to satisfy an ascending chain condition. This condition, that for every ascending chain $s_0 < s_1 < \ldots < s_k$ some $f(s_k) = f(s_{k+1})$, also guarantees that evaluation of each individual SCC will terminate, and thus that the entire evaluation process will terminate.

While there are application areas whose natural semantic functions satisfy one of these conditions, there are also some natural functions that are not so well-behaved. An example of such functions arises when we try to use attribute evaluation to model execution of a while loop. Adding an indefinitely looping construct to model while execution will give the attribute evaluation mechanism full partial recursive power to apply to other, less easily visualized problems. We consider the small grammar found in Figure 2.1 to be a natural way to express interpretation via attribute evaluation. (A value of not-reached for a state means that the corresponding statement would not be executed if the program were run.)

```
Start :: = S
    S.initst = λname.⊥
    Start.finalst = S.finalst
S :: = id ← exp
    exp.state = S.initst
    S.finalst = if S.initst = not-reached then
                    not-reached
                else
                    S.initst[id.name←exp.val]
exp :: = id
    exp.val = lookup(id.name,exp.state)
exp :: = int
    exp.val = int.val
exp1 :: = exp2 op exp3
    exp2.state = exp1.state
    exp3.state = exp1.state
    exp1.val = exp2.val op exp3.val
S1 :: = S2 ; S3
    S2.initst = S1.initst
    S3.initst = S2.finalst
    S1.finalst = S3.finalst
S1 :: = while exp do S2
    while.gate = if while.start then
                    ⟨1,S1.initst⟩ (initial value)
                else
                    ⟨0,S2.finalst⟩ (subsequently)
    exp.state = second(while.gate)
    S2.initst = if first(while.gate) = 1 and
                    exp.val = 0 then
                    not-reached
                else if exp.val ≠ 0 then
                    second(while.gate)
                else
                    S2.initst
    S1.finalst = if exp.val = 0 then
                    S2.finalst
                else
                    not-reached
```

Figure 2.1. A gated attribute grammar.

If the **while** production is ignored, this attribute grammar is acyclic. With the **while** production, a cycle is formed by the while.gate, B.state, B.val, S_2.initst, and S_2.finalst attributes, as shown in Figure 2.2, where dashed links indicate indirect dependencies. (The while.start attribute will be discussed later.)

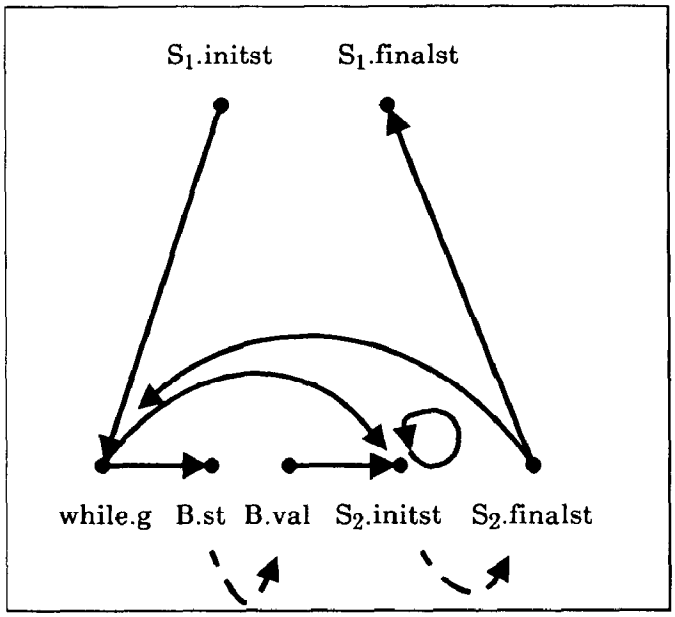

Figure 2.2. **while** dependencies.

For the class of problems and algorithms considered here, it is appropriate to use the standard total ordering on the integers, rather than creating a flat cpo out of the integers. (Circular attribute grammars give rise to sets of simultaneous equations over the integers, and, as expanded upon in the next section, solutions of such systems with respect to the flat cpo would give the value $\perp$ to many of the variables.) Since there is no reason, under this ordering, to expect that the body of the **while** loop represents a monotonic function from initial states to final states, the previous work on circular attribute grammars can not define the desired final state of this loop. Execution of this loop, however, would result in a well-defined final state, provided that the loop terminates. In the next section, we extend the usual attribute evaluation algorithm in such a way that the attribution final state coincides with the execution final state.

This extended attribution algorithm does not require an attribute grammar to be noncircular, but it still requires some order in the attribute grammar. Every nontrivial dependency-graph cycle that can be generated by such an attribute grammar must contain at least one attribute designated as a *gate*. (Cycles such as the one for S_2.initst in Figure 2.1 with only one node in the cycle, corresponding to an attribute depending on its prior value, are considered trivial.) A gate attribute represents the point at which attribution of a strongly-connected component in the dependency graph should begin. Each gate attribute has an automatically associated *start* attribute which will indicate when the gate attribute should take its value from outside its SCC rather than inside. In the example, while.gate is the only designated gate and while.start is its start attribute.

A gate attribute provides the means of controlling the evaluation of attributes within a strongly connected component. If all functions in the SCC are monotonic and all attributes are initially $\perp$, starting propagation from any attribute in the SCC will yield the least fixed point of the SCC; if some functions are nonmonotonic, starting propagation from different attributes can yield different fixed points, so we have defined the desired fixed point to be the one where propagation starts at the gate attribute. By guaranteeing that each SCC has a gate attribute, we guarantee that each SCC has a well-defined fixed point.

Given an attribute grammar with designated gate attributes, the algorithms previously used to detect circularities can now be used with only minor modifications to detect circularities that do not pass through gate attributes. If no such circularities can be found, the attribute grammar is called a *gated*

attribute grammar and can be evaluated with our techniques.

Each gate in a dependency graph defines a *gated SCC*, which can be considered as the region of the graph under the control of the gate. Those attributes used to calculate the gate value when its start is true are outside the gate's control, so edges from them to the gate are not used in determining the extent of the GSCC. Any node that remains strongly connected to the gate without these edges in the graph is in the gate's GSCC.

As a consequence of this definition, every nontrivial maximal SCC in a dependency graph of a gated attribute grammar is a GSCC and, for any gate which is not nested inside another's GSCC, the GSCC is the maximal SCC containing that gate.

The *sub-GSCC* for an interior gate node consists of those nodes in the SCC that are strongly connected to the interior gate after removing arcs to this gate from its start node and whichever other nodes contribute to the gate value when its start is true. This sub-GSCC is the same as the maximal SCC for the interior gate node would be if there were not an outer gate. The portion of a GSCC which is not in any sub-GSCC is called its *core*.

3. Initial Evaluation.

Initial evaluation starts with an attributed parse tree where no attributes are guaranteed to have consistent values. Therefore, every attribute needs to be evaluated, those in nontrivial SCCs possibly more than once, for the tree to become consistently attributed.

A parse tree is attributed by repeatedly selecting a strongly connected component (potentially a single attribute) of its dependency graph and evaluating the attributes in that SCC until they reach final consistent values. SCCs can be selected in any way consistent with a topological ordering of the collapsed dependency graph, where each maximal SCC has been reduced to a single node. Thus, if there are no nontrivial SCCs in the graph, our evaluation process proceeds in the same way as the usual optimal evaluation process.

Before evaluation of the attributes mentioned by the grammar writer begins, certain additional links are added to the dependency graph by nonlocal productions. These additional links, indicated by dotted lines in Figure 3.1, make each attribute (outside a GSCC) that is a predecessor of a node in the GSCC also a predecessor of the start attribute of that GSCC. These links allow the evaluator to always begin evaluation of a GSCC at its gate attribute. The evaluation function for a start attribute returns true, indicating that the gate attribute should take its value from outside the GSCC to begin GSCC evaluation, iff any of its predecessors from these additional links were newly set or changed value since the gate attribute last took a value from outside. Since the only successor of a start attribute is its gate attribute, adding these additional links does not change the SCC composition of the dependency graph.

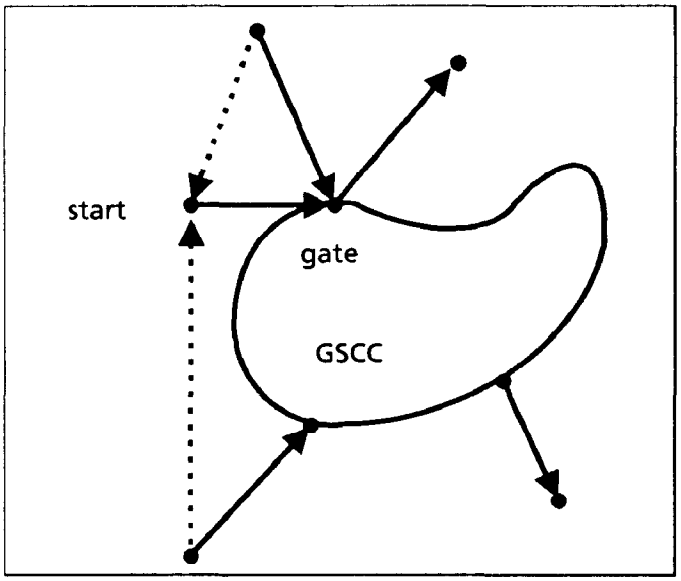

Figure 3.1. Adding non-local links.

These links are useful primarily for incremental re-evaluation of a GSCC, but even during initial evaluation sub-GSCCs may require multiple evaluations,

and any GSCC may require multiple passes to reach a fixed point. In practice, subsequent passes and evaluations of a GSCC would be handled by the more efficient incremental evaluation of the next section, but for the moment we consider a simplified algorithm where all the attributes of a GSCC are evaluated on each pass.

Since there may be nonmonotonic functions inside a GSCC, attribute evaluation order influences correctness as well as efficiency. An example where differing evaluation order results in different fixed points for the GSCC is shown in Figure 3.2. The attributes are shown with consistent values that could be left over from an earlier evaluation of the GSCC. Now assume that the GSCC must be evaluated again, and that the gate g gets the new initial value 20. If the GSCC is evaluated in the order acbgabcg, the fixed point (a:5, b:-5, c:10, g:15) is reached. If it is evaluated in the order abcg, the fixed point (a:10, b:-10, c:20, g:20) is reached.

This example also justifies our choice of ordering for the integers. Using a flat cpo, we would find that the least fixed point of this GSCC is (a:⊥,b:⊥,c:⊥,g:⊥). While this is a solution to the set of simultaneous equations representing the GSCC, we believe the writer of the attribute grammar would prefer the integer solution (a:10, b:-10,c:20,g:20). Similarly, if the cpo for states is constructed from the flat cpo for the integers in order to guarantee monotonicity of functions, the least fixed point of a **while** loop would have each attribute in the SCC at its bottom value, meaning that the final state of every **while** loop would have no identifiers defined.

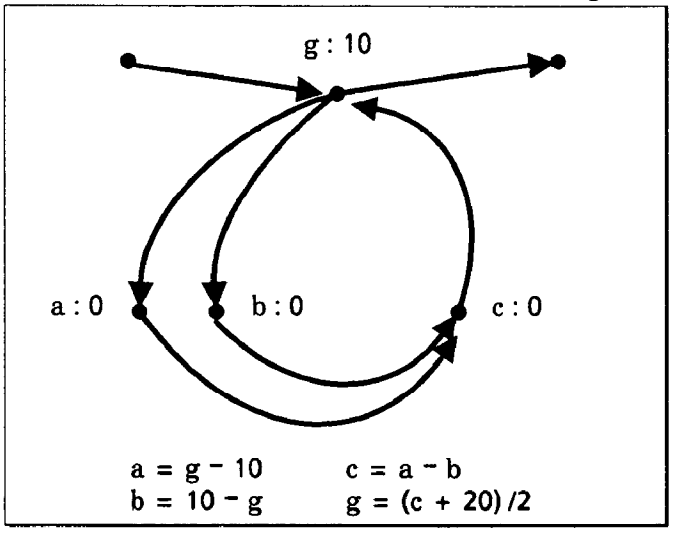

Figure 3.2. A nonmonotonic GSCC.

We define the correct evaluation order within a GSCC by a topological ordering on the GSCC graph with edges leading to the gate removed (and any sub-GSCCs collapsed to single nodes). In other words, an attribute in the interior of a GSCC cannot be evaluated unless all attributes on paths from the gate to it are consistent with the present gate value. There are cases, such as Figure 3.3, where such an ordering cannot be obtained due to a cycle interior to a GSCC, but such cycles are ruled out in gated attribute grammars by the requirement that each nontrivial cycle contain a gate attribute.

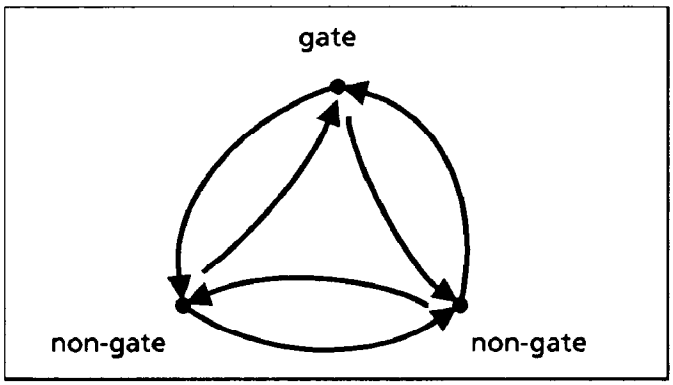

Figure 3.3. Inevaluable GSCC.

A nontrivial GSCC is evaluated by alternating between evaluating the gate and evaluating the other nodes of the GSCC in internal topological order, with possible recursive evaluations of sub-GSCCs, until one pass is made over the GSCC with no changes in attribute values.

Even with monotonic functions, [JoS86] for efficiency restrict evaluation order to agree with a depth-first ordering starting at a defined set of SCC

attributes. For the first pass over the SCC, the set consists of those SCC attributes whose predecessors outside the SCC have changed. On succeeding passes, the set consists of the successors of leaves of the depth-first spanning tree from the previous pass. Since all their functions are monotonic, the evaluation ordering from these spanning trees could be replaced by one derived from a topological ordering on the SCC with incoming arcs to the defined set temporarily deleted. With such a replacement, the defined set for each pass would be a subset of that for the previous pass, with nodes being dropped from the set if their predecessors did not change on the previous pass. Our algorithm starts with only the gate attribute in the defined set, and SCC evaluation terminates when the gate's predecessors in the SCC do not change on a pass.

4. Incremental Evaluation.

At the beginning of incremental evaluation, a parse tree is consistently attributed except for a small number of attributes corresponding to changes that have just been made in the parse tree. The strongly connected components containing these attributes are placed in the set NeedSCCEval, which throughout incremental evaluation contains those SCCs known to need evaluation. Another set, NeedEval(S), which contains those attributes of the GSCC S that are known to need evaluation, is kept for each GSCC (i.e. nontrivial SCC) in NeedSCCEval. While NeedSCCEval is not empty, an SCC is removed and evaluated. SCCs are selected for evaluation in accordance with the topological order imposed by the collapsed dependency graph.

A trivial SCC is evaluated by evaluating its node (repeatedly if the SCC is a trivial cycle) and adding its SCC-successors to NeedSCCEval if the value of the node changed. A nontrivial SCC is evaluated in the same spirit, as shown in Figure 4.1. Starting from the gate, changes are propagated around the GSCC

```
for each node n in S with a successor outside S,
   save the present value of n
evaluate gate(S), setting start(S) false
if gate(S) changed, add successors of gate(S) in S
   to NeedEval(S)
while NeedEval(S) ∩ core(S) is not empty
   select n from NeedEval(S) ∩ core(S) in
      accordance with internal topological order
   if n is a start node,
      evaluate n
      if n is true, recursively evaluate the associated
         sub-GSCC, T
      add successors of T in S-T to NeedEval(S)
   else
      evaluate n
      if n changed,
         add successors of n in S to NeedEval(S)
for each node n in S with a successor outside S,
   if its present value differs from its saved value,
      add its SCC-successors to NeedSCCEval
      add its successors to NeedEval sets
```

Figure 4.1. Evaluating the non-trivial SCC S.

in internal topological order until attributes settle to consistent values. Then SCC-successors are added to NeedSCCEval if their predecessor attributes have values different from before SCC evaluation.

Evaluating a sub-GSCC is identical to evaluating a GSCC, except that old attribute values are not saved and checked to add SCC-successors, since a sub-GSCC cannot know if its attribute values are final when they settle to consistent values. The sub-GSCC may be re-evaluated, changing its attribute values further, in the process of evaluating the GSCC as a whole.

Whether evaluating a GSCC or a sub-GSCC, it suffices to evaluate attributes that are in the core and are known to need evaluation. If an attribute in a further nested GSCC needs evaluation, so does the start attribute of that nested GSCC, which is in the core of the next outer GSCC. Evaluating that start attribute will trigger evaluation of the nested GSCC, including the needy attribute.

Also note that the NeedEval set of the maximal GSCC can be used by the sub-GSCC evaluation, which will look only for attributes in the core of the sub-GSCC. Thus it is not necessary to worry about multiple NeedEval sets for different parts of the same maximal SCC.

5. Avoiding Re-evaluation of Loops.

Often when an input to a loop changes, it is necessary to entirely re-evaluate the loop. Where possible, however, we would like to avoid this expense. The basic idea, that of identifying portions of a loop that do not affect the computation of the loop in such a way as to require an entire re-evaluation, is reminiscent of code hoisting in optimization.

This discussion is necessarily specific to our example of an attribute grammar based interpreter in which attributes containing states are passed around the tree.

Just as in standard incremental evaluation, we hope that at some point short of re-evaluating all the attributes in the tree either attributes stop changing value or they change value in such a way as not to require further re-evaluation. To make this determination it is useful to define *input* and *output* variables of a loop. An input variable is one whose value is queried somewhere inside the loop, and an output variable is one whose value is modified somewhere inside the loop. Variables may of course be both input and output variables for a given loop. For the example language of this paper, the input variables are those that appear on the right-hand side of an instance of the production exp :: = id and the output variables are those that appear as left-hand sides of assignment statements. We extend the concept of the state value not-reached to include an extra pair of integers in each state attribute. These integers will record the number of the loop iteration at which the state is first evaluated and the number of the loop iteration at which it is last evaluated.

During change propagation, say we find that a predecessor of a state-valued attribute in a GSCC corresponding to a loop has changed. We evaluate the state inside the GSCC and note which variables inside it change value. We consider each changed variable in turn. The first iteration in the loop during which the state was evaluated represents the first time that the variable being considered would actually receive the new value. If that is after the last use (if any) of the variable in the loop, then we do not need to re-evaluate the loop; we need only update those states evaluated after the given one to reflect the new "final value" of the variable. (Variables that are independent of the loop are handled similarly.)

Changing the "clean-up" code in a loop such as that in Figure 5.1 intuitively should not require re-evaluation of the entire loop, and the above relatively inexpensive technique formalizes and captures such situations.

```
while not done do
    { main loop body }
    if condition then
        { clean-up code };
        done : = true
    else
        { continue iterating }
    fi
od
```

Figure 5.1. Typical loop.

6. Nonlocal Predecessors of Start Attributes.

It is necessary to create links from predecessors of a GSCC to the start attribute of that GSCC. These links are used to make the start attribute true and schedule the GSCC for re-evaluation whenever an input to the GSCC changes value in such a way as to require a full re-evaluation of the GSCC. We use superior and inferior characteristic graphs as necessary to find attributes that are in a GSCC and

identify predecessors that are not in the GSCC. The first step in creating the needed links is to associate a pointer to the appropriate gate attribute with each member of a GSCC. In the case of nested GSCCs, each member gets a pointer to the gate of the smallest containing GSCC. In this way, all exterior gates can be found by following the chain of gate pointers.

```
Start ::= S
    S.isgp = nil
S1 ::= while exp do S2
    while.gategp = S1.isgp
    S1.fsgp = S1.isgp
    S2.isgp = @while.gate
    exp.sgp = if exp.infchar = ⟨state⇒val⟩ then
                  @while.gate
              else nil
S ::= id ← exp
    S.fsgp = S.isgp
    exp.sgp = if exp.infchar = ⟨state⇒val⟩ then
                  S.isgp
              else nil
S1 ::= S2 ; S3
    S1.fsgp = S1.isgp
    S2.isgp = S1.isgp
    S3.isgp = S1.isgp
exp ::= id
    exp.valgp = exp.sgp
exp ::= int
    exp.valgp = exp.sgp
exp1 ::= exp2 op exp3
    exp2.sgp = if exp2.infchar = ⟨state⇒val⟩ then
                   exp1.sgp
               else nil
    exp3.sgp = if exp3.infchar = ⟨state⇒val⟩ then
                   exp1.sgp
               else nil
    exp1.valgp = exp1.sgp
```

Figure 6.1. Added flow rules.

For our example grammar, we calculate gate pointers by adding the evaluation functions in Figure 6.1. (The notation $@X.a$ indicates that a pointer to attribute X.a is produced.) Statement nonterminals have initial state and final state attributes, and the associated gate pointer attributes are abbreviated isgp and fsgp respectively. A similar naming convention is used for gate pointers associated with state and value attributes of expression nonterminals and gate attributes for **while** symbols.

Once we have the gate pointers, we can add the necessary links. An attribute is a predecessor of a GSCC if the gate pointer of a successor does not appear in the chain of gate pointers from the attribute (*i.e.*, the successor is not in a containing GSCC). Since the successor may be in several nested GSCCs, a link is added from the predecessor attribute to the start pseudo-attribute associated with each gate pointer in the trace-back chain from the successor until this trace-back chain joins the one from the predecessor. Figure 6.2 shows a parse tree with the links added after execution of this first phase of attribute evaluation.

7. Conclusions.

Like [JoS86], we guarantee that each maximal SCC evaluated will be in AFFECTEDSCC and that no SCC will be evaluated more than once in a single incremental evaluation. However, due to the lack of restriction on the functions within an SCC, we cannot bound the number of attribute evaluations required to evaluate a single SCC. To gain the power to model execution of a **while** loop in attributes, we are forced to accept that evaluation of an SCC may not terminate if its corresponding **while** loop diverges. (Evaluation of an SCC representing a divergent **while** loop will in fact terminate if the states in the **while** loop stop changing, as in the loop **while** 1 **do** i ← 2. In such a case, any subsequent states in the program are given the not-reached value.)

The recursive synth-function evaluator described in [Far86] addresses evaluation process termination problems in recursive attribute grammars by pulling the evaluation of an SCC into a single function and

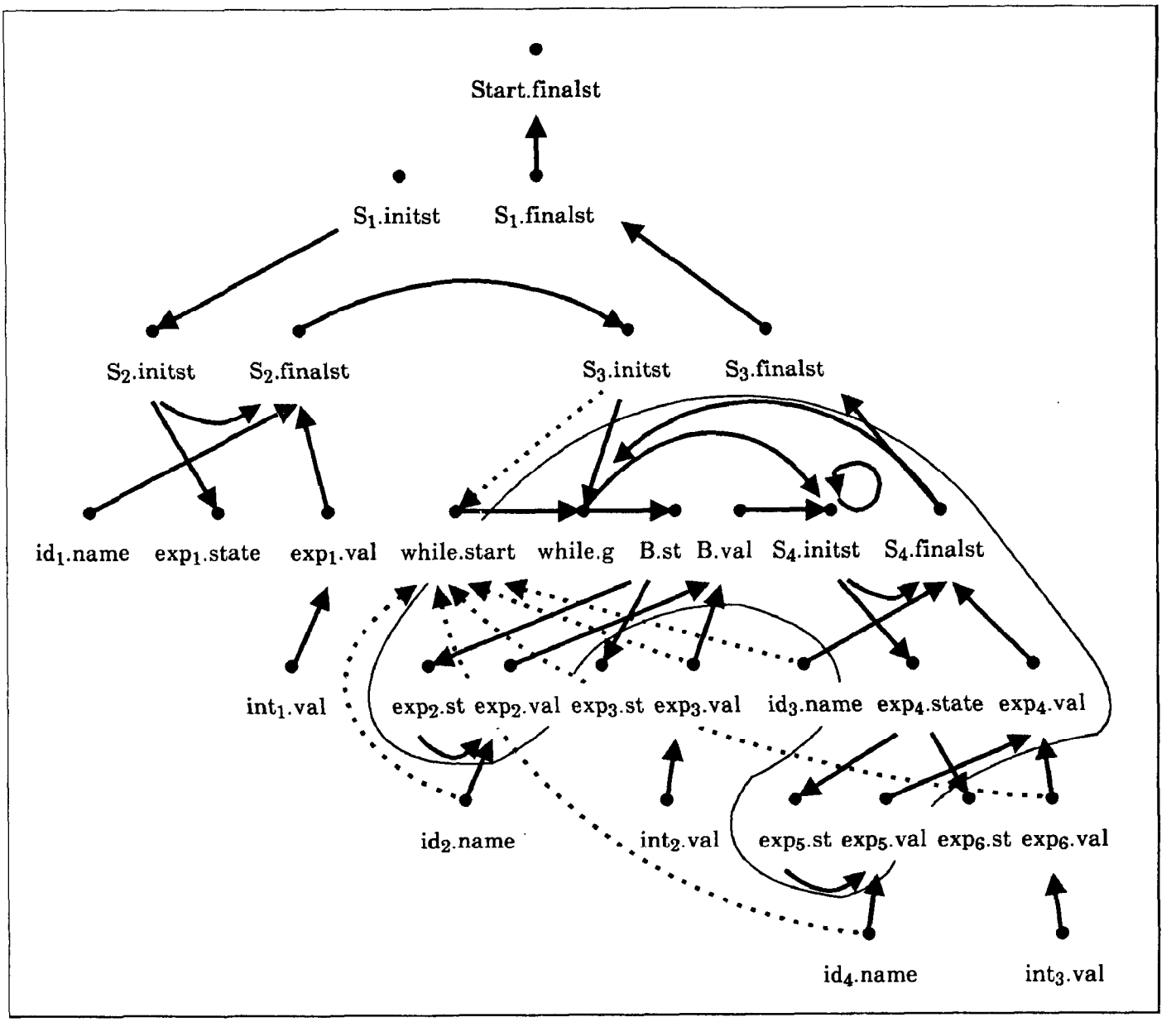

Figure 6.2. Dependency Graph for i ←10 ; while i > 0 do i←i-1.

then pointing out that only a finite number of function evaluations will be attempted. For his finitely recursive attribute grammars, this transformation does not generate partial evaluation functions unless some of the SCC functions were partial. (Most functions in traditional attribute grammars are simple total functions, but there is no way in general for an evaluator generator to guarantee these functions will always terminate.) However, for general circular attribute grammars this transformation would not affect the actual termination of the evaluation process, but would merely move potential nontermination from the attribute selection phase to the attribute evaluation phase.

We have thus allowed partial functions to be explicitly spread over the computation of the attributes of a GSCC instead of restricted to the computation of a single attribute. This choice allows computation to be expressed via attribute

evaluation functions in whichever way seems most convenient.

By forcing attribute evaluation to occur in accordance with a topological ordering of the dependency graph, we guarantee that attribute evaluation will terminate if interpretation of the corresponding program would terminate. In addition to allowing a GSCC to converge to the wrong fixed point, evaluation out of topological order can result in undesired nontermination. If a GSCC were to be evaluated when some of its predecessors had yet to reach their final values, the invalid combination of inputs might be such that the GSCC has no reachable fixed point. Yet GSCC evaluation cannot stop before it reaches a fixed point, for on valid input the evaluation must continue until it determines that the corresponding loop converges. However, topological ordering for the sake of efficiency will also guarantee that all GSCC inputs are valid when the GSCC is evaluated.

To gain the ability to evaluate arbitrary functions in dependency graph cycles, we have been forced to accept one localized timing constraint on attribute evaluation. A start attribute is required to know if any of its predecessors have changed *since the last time its gate attribute took a value from outside the GSCC*. This constraint is met by setting the start attribute false immediately after, but conceptually at the same time as, taking a gate value from outside the GSCC. Without some way of telling whether GSCC evaluation is just beginning, a gate cannot know which of its evaluation functions to use. Using the internal evaluation function at the beginning of GSCC evaluation may abruptly end that evaluation if the gate is the only GSCC attribute with modified predecessors; using the external function in the midst of GSCC evaluation again brings up the problem of possible divergence due to invalid combinations of values. [JoS86] face a somewhat similar problem in incrementally evaluating SCCs with monotonic functions. They must keep track of whether an attribute has been evaluated in the current round of SCC traversal to ensure that a left-over value does not cause SCC evaluation to miss its new least fixed point.

With the use of an efficient representation, such as may be found in [Hoo87], for aggregate attributes like states that differ only slightly from one another, the performance penalty for propagating state information around a dependency graph may be substantially reduced from that of a naive implementation. It is our belief that any remaining penalty is more than offset by the advantage of being able to automatically start re-execution at the point of change in a program.

References.

[BaS86] Bahlke, R. and G. Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Transactions on Programming Languages and Systems* **8**(4), pp. 547-576 (October 1986).

[BMS87] Bahlke, Rolf, Bernhard Moritz, and Gregor Snelting, "A Generator for Language-Specific Debugging Systems," *Proc. of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices* **22**(7), pp. 92-101 (June 1987).

[Far86] Farrow, Rodney, "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars," *Proc. of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices* **21**(7), pp. 85-98 (July 1986).

[FJM83] Fischer, C.N., G.F. Johnson, J. Mauney, A. Pal, D.L. Stock, "An Introduction to Editor Allan POE," *Proc. of Softfair, A Conference on Software Development Tools, Techniques, and Alternatives* (July 1983).

[Hoo86] Hoover, Roger, "Dynamically Bypassing Copy Rule Chains in Attribute Grammars," *Proc. of the Thirteenth ACM Symposium on Principles of Program-*

ming Languages, pp. 14-25 (January 1986).

[Hoo87] Hoover, Roger, *Incremental Graph Evaluation*, Ph.D. thesis, TR 87-836, Cornell University (May 1987).

[JoF85] Johnson, Gregory F. and C. N. Fischer, "A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors," *Proc. of the Twelfth ACM Symposium on Principles of Programming Languages*, pp. 141-151 (January 1985).

[JoS86] Jones, Larry G. and Janos Simon, "Hierarchical VLSI Design Systems Based on Attribute Grammars," *Proc. of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 58-69 (January 1986).

[KaW87] Karinthi, Raghu R. and Mark Weiser, "Incremental Re-Execution of Programs," *Proc. of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, *SIGPLAN Notices* **22**(7), pp. 38-44 (June 1987).

[Kas80] Kastens, Uwe, "Ordered Attribute Grammars," *Acta Informatica* **13**, pp. 229-256.

[Knu68] Knuth, Donald E., "Semantics of Context-free Languages," *Mathematical Systems Theory* **2**(2), pp. 127-145 (June 1968). Correction **5**(1), pp. 95-96 (March 1971).

[ReT84] Reps, Thomas and Tim Teitelbaum, "The Synthesizer Generator," *Proc. of the ACM Software Engineering Symposium on Practical Software Development Environments*, pp. 42-48 (April 1984).

[RMT86] Reps, Thomas, Carla Marceau, and Tim Teitelbaum, "Remote Attribute Updating for Language-Based Editors," *Proc. of the Thirteenth ACM Symposium on Principles of Programming Languages*, pp. 1-13 (January 1986).

[RTD83] Reps, Thomas, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Transactions on Programming Languages and Systems* **5**(3), pp. 449-477 (July 1983).

[Ske78] Skedzeleski, Stephen K., *Definition and Use of Attribute Reevaluation in Attributed Grammars*, Ph.D. thesis, University of Wisconsin (December 1978).