Concurrent Incremental Attribute Evaluation

Henk Alblas

University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: alblas@cs.utwente.nl

Abstract. The design of a concurrent incremental combined static/dynamic attribute evaluator is presented. The static part is an incremental version of the ordered attribute evaluation scheme. The dynamic part is an incremental version of the dynamic evaluation scheme.

To remove the restriction that every transformation of an attributed syntax tree should immediately be followed by a reevaluation of the tree, criteria have been formulated which permit a delay in calling the reevaluator. These criteria allow multiple asynchronous tree transformations and multiple asynchronous reevaluations. Transformation and reevaluation processes are distributed over regions of the tree. Each region is either in its transformation phase or in its reevaluation phase. Different regions can be in different phases at the same time.

1. Parallel compilation

Compilers are among the tools most heavily used by programmers. Therefore, the speed of compilation may have a great impact on programmer productivity. One way to speed up compilation is the application of parallelism. Other techniques to reduce compile time are incremental compilation and separate compilation of program modules. Parallel compilation can make both of these techniques faster. As parallel processing technology advances, concurrency becomes an attractive vehicle by which compilation speed may be increased.

Any practical investigations in concurrent compilation have naturally been driven by the multiprocessing hardware available at the time. Early efforts were aimed at the application of vector processing techniques, while more recent research has been directed towards more coarsely grained parallel compiler designs. The easiest way in which to transform a sequential compiler into a concurrent compiler is to run the compiler phases as separate processes in a pipeline, and to have them communicate through shared data structures. This approach has two limitations. First, the degree of concurrency is limited by the number of stages in the pipeline. Second, the overall speed of compilation is constrained by the speed of the slowest stage.

Instead of determining which phases of the compilation process can be carried out concurrently, one may construct a compiler which splits the source program into segments which are then compiled concurrently. These segments would typically be well-defined syntactic constructs, such as procedure bodies or statement lists. Lipkie [16] was probably the first to suggest a combination of pipelining and concurrent processing of program segments. He proposed the division of the source program at procedure boundaries and to instantiate a pipeline for each procedure. Frankel [8] extended a one-pass recursive descent Pascal compiler to translate procedures concurrently. Whenever a compiler instance encounters a child scope while compiling a parent scope, it creates a new instance to compile the child scope and skips to the end of the child scope by matching delimiters. Vandevoorde [23] constructed a concurrent C compiler that consists of a two stage pipeline. The first stage performs extended lexical analysis for the second stage, which does the parsing, semantic analysis and code generation. The second stage is able to process units as small as simple statements concurrently. To dynamically restrict unproductive concurrency, a scheduling strategy is used which favours serial execution when parallel execution is unproductive and favours coarser grains of parallelism over finer ones. Seshadri et al. [20, 21, 22] used a similar approach to build parallel Modula 2+ compilers to run on a distributed system. They do not restrict themselves to the parallel compilation of statements, but they investigate the concurrent processing of declarations as well. The latter introduces what they call the "doesn't know yet" problem: the compiler may attempt to access a variable before its declaration has been processed.

Theoretical research in the field of parallel compilation has dealt mainly with its best understood area, namely parsing regular and context-free grammars. Attribute grammars have proved to be a useful formalism for specifying the context-sensitive syntax and the semantics of programming languages, as well as for implementing editors, compilers, translator writing systems and compiler generators. Therefore, the theoretical investigation into parallel attribute evaluation also deserves attention.

The rest of this paper is devoted to parallel attribute evaluation, and in particular, parallel incremental attribute evaluation. In Section 2 we recall the notions static and dynamic attribute evaluation, and further distinguish between non-incremental and incremental evaluation, and between sequential and parallel evaluation. In Section 3 we consider possible distributions of attributes over processes. In Section 4 we summarize the parallel evaluation strategies that are known from the literature. Conditional tree transformations are defined in Section 5 to motivate incremental attribute evaluation. In Section 6 we present a new approach to concurrent incremental attribute evaluation, which combines both static and dynamic evaluation strategies. In Section 7 we extend our approach to multiple asynchronous tree transformations, and formulate criteria which permit a delay in calling the reevaluator. This requires a different view of the correctness of attribute values of a syntax tree. The use of approximate attribute values, as defined in [4], allows multiple asynchronous tree transformations and multiple asynchronous reevaluations.

2. Attribute evaluation strategies

An *attribute evaluator* computes the values of all attribute instances attached to a syntax tree. The constraint that an evaluation rule can only be executed when the values of its arguments are available, naturally leads to the concept of dependency graphs. For every syntax tree T a *dependency graph* $D(T)$ can be defined by taking the attribute instances of T as its vertices. The directed arc (a, b) is contained in the graph if and only if attribute instance b depends on attribute instance a, i.e. if a is an argument in the evaluation rule for b. Thus, the existence of arc (a, b) indicates that the value of a must be available before b can be computed. We restrict our attention to attribute grammars for which dependency graphs are acyclic.

2.1 Static and dynamic evaluators

Traditional (i.e., sequential) attribute evaluation methods can be divided into two categories: *dynamic* and *static* evaluation. Given a source program and its associated syntax tree T, a dynamic evaluator constructs the dependency graph $D(T)$ and performs a topological sort of the nodes in $D(T)$ to determine the evaluation order of the attribute instances in T. A static evaluator does not construct the dependency graph. Instead, the evaluation order is based on the dependencies between the attribute occurrences in the productions of the attribute grammar. As a result, a static evaluator can be used for any tree of the grammar. The evaluation order of a dynamic evaluator is determined at attribute evaluation time, whereas the evaluation order of a static evaluator is determined at evaluator generation time. On sequential machines, static evaluators are preferred because they are more efficient in time and space. Dynamic evaluators, however, have a higher potential for concurrency.

2.2 Non-incremental and incremental evaluators

We can also distinguish between *non-incremental* and *incremental* attribute evaluators. A non-incremental attribute evaluator computes the attribute instances attached to the nodes of an invariant syntax tree only once. A tree transformation may invalidate attribute instances, not only in the restructured part of the tree but (because of long reaching attribute dependencies) also elsewhere in the tree. To make the attribution of the tree correct again, a reevaluator must be activated. However, a repeated computation of all the attribute instances after each transformation is inefficient and should be avoided. The only attribute instances subject to reevaluation are the attribute instances having an incorrect value and the attribute instances directly depending on these attribute instances. Algorithms exist that work optimally in the number of visits to tree nodes and in the number of recomputations. Evaluators employing such algorithms are called incremental evaluators.

2.3 Sequential and parallel evaluators

Yet another distinction is between *sequential* and *parallel* evaluators. For a given syntax tree *T* the evaluation order of its attribute instances is only restricted by the partial order induced by the associated dependency graph *D*(*T*). A sequential attribute evaluator completely serializes this order. In a parallel attribute evaluator the partial order does not have to be serialized. Instead, attribute computations are allocated to evaluation processes. As in the case of sequential evaluators, the way in which this is done can be determined at evaluator generation time or at attribute evaluation time.

3. Distributions of attributes over processes

Attribute instances of a syntax tree can be distributed over processes in several ways. Kuiper [15] distinguishes between *tree-based distributions*, *attribute-based distributions* and a combination of these two distributions, called *combined distributions*.

A distribution is called tree-based if all attribute instances attached to the same tree node are assigned to the same evaluation process. Generally, the tree is divided into *connected parts*, called *regions*. Each evaluation process evaluates attribute instances in a region of the tree. To define a region it suffices to define which node is the common ancestor of the nodes in the region. Criteria to select common ancestors are the production applied at a node (*production-based distribution*) and the nonterminal labelling a node (*nonterminal-based distribution*). Furthermore, *nested* and *non-nested distributions* can be distinguished. In the case of nested distributions the production or the nonterminal is the only selection criterion. In the case of non-nested distributions an additional criterion has to be taken into account, namely that a node is selected if and only if none of its ancestors is selected.

A distribution is called attribute-based if a process is allocated to all instances of an attribute or to all instances of an attribute occurrence. An example of an attribute based distribution is the association of a process with each evaluation pass of a multi-pass evaluator.

A combined distribution is the result of a tree-based distribution followed by an attribute-base distribution. The tree is divided into regions according to the tree-based distribution, and then for each region, processes are allocated to attribute instances according to the attribute based distribution.

4. Parallel attribute evaluation

For a given syntax tree the evaluation order of its attribute instances is only restricted by the partial order induced by its dependency graph. Therefore, an evaluator which completely exploits the potential for concurrency should be based on partial orders only. Parallelizing the dynamic evaluation scheme is straightforward. Each evaluator builds the dependency graph for its region, marking attribute instances to be computed in other regions as unavailable. Next, the evaluator does a topological sort of its attribute instances, and starts evaluation. An evaluator may have to wait for attribute instances from other regions. When they become available, the dependency graph is updated accordingly. Additionally, each of the evaluators has to make attribute instances available to other evaluators. Although this method achieves a high degree of concurrency, one must realize that the analysis of dependency graphs as part of the attribute evaluation process to detect possible parallelism may require more time than it saves. Instead, the analysis of an attribute grammar at evaluator generation time to detect potential parallelism can be more elaborate and more time consuming because it is performed only once for the grammar and can be applied to any syntax tree.

Unfortunately, all known static evaluation strategies completely serialize the evaluation process by introducing additional dependencies. Sequential thought patterns seem to hinder people in inventing parallel static evaluation methods. The thesis of Schell [19] includes a modest start to a theoretical investigation into parallel tree-walk evaluation. Whenever possible, nodes are visited concurrently by assigning separate processes to traverse subtrees rooted at the same node. These processes are forked in a top-down descent of the derivation tree. Kuiper [15] presents an algorithm that analyses attribute grammars in order to statically (i.e., at evaluator generation time) detect independent computations. He also presents a method that increases the amount of potential parallelism by removing attribute dependencies and thus cutting linear chains at the cost of adding extra attributes. To date, however, no parallel static attribute evaluation classes have been invented.

The dilemma is that, on one side, static evaluators are efficient but hard to parallelize, and on the other side, parallel dynamic evaluators are easy to construct but impractical to use. To solve this dilemma, Böhm and Zwaenepoel [6] designed a combined static/dynamic evaluator which tries to combine the potential for concurrency of dynamic evaluators with the sequential efficiency of static evaluators. Their static evaluation method is the ordered attribute grammar scheme [13], and their dynamic evaluation method is based on the availability of attribute instances in the polynomial sort of the dependency graph.

The structure of their parallel compiler is roughly as follows. The sequential parser builds the syntax tree, divides it into non-nested "bottom" subtrees, and a remaining "top" tree, and sends them to the attribute evaluators. The attribute evaluators then proceed with the actual translation by evaluating attribute instances belonging to the symbols in their region, transmitting values of shared attribute instances as necessary. Dynamic evaluation is applied to the top tree, whereas all bottom subtrees are evaluated entirely statically. The distribution is such that the vast majority of attribute instances is evaluated statically. The children of each tree node N of the top tree are inspected to see if any of them should be handled by a static evaluator. If so, the (transitive) *inherited*-to-*synthesized* dependencies between the child's attribute instances (as precomputed by the static evaluator) are entered into the dynamic dependency graph, as are the dependencies induced by the evaluation rules associated with the production applied at N. When the tree construction is completed, evaluation starts in topological order, as for dynamic evaluators. When all predecessors for a statically evaluated attribute instance become available, the appropriate static visit procedure is invoked. In reverse, a static evaluator may activate the dynamic evaluator. To every static subtree evaluator a process is assigned. One could assign several processes to the dynamic top tree evaluator (e.g., one process to every attribute instance that is ready for evaluation), but one process suffices if it can keep the majority of the static evaluators busy in parallel with its own activities.

In both the static and the dynamic evaluator each attribute instance is computed once, and only after its arguments have received their correct value. The same holds for the combined static/dynamic evaluator. Hence, it is guaranteed that the value of every attribute instance is correct after the completion of the evaluation process.

In what precedes we assumed the syntax tree remains unchanged as far as attribute evaluation is concerned. In the following sections we consider the problem of attributed syntax trees which may change during the compilation process. Therefore, we first describe the concept of attributed tree transformations and then present a design for a parallel incremental evaluator.

Although we will discuss incremental attribute evaluation in the context of optimizing tree transformations, our method can be applied to any syntax tree with incorrect attribute values. Hence, our incremental evaluator works for incremental editing as well.

5. Conditional tree transformations

Both compiler optimizations and (context-sensitive) syntax-directed editing can be described by tree transformations. To specify optimizing tree transformations the classical attribute grammar framework has to be extended with conditional tree transformation rules, where predicates on attribute values (carrying context information) may enable the application of a transformation (see e.g. [24]). We restrict ourselves to tree transformations which preserve the syntax, i.e., all intermediate trees are syntax trees in the same context-free grammar. A tree transformation rule consists of an input tree template *itt*, and an output tree template *ott* (for reasons of simplicity we assume that *itt* and *ott* have equally labelled roots). Context conditions can be expressed by *enabling conditions* which are predicates on attribute instances of *itt*.

The set of attribute instances of a tree template can be naturally partitioned into three disjoint subsets of input, output and inner attribute instances. The *input attribute instances* are the inherited attribute instances of the root and the synthesized attribute instances of the leaves; the *output attribute instances* are the synthesized attribute instances of the root and the inherited attribute instances of the leaves; the *inner attribute instances* are the attribute instances of the inner nodes.

The inner and output attribute instances of *ott* are completely determined by the input attribute instances of *ott* and the ordinary evaluation rules associated with the productions applied in *ott*. It is assumed that corresponding input attribute instances of *itt* and *ott* keep their values. Explicit evaluation rules are needed, however, for the synthesized attribute instances associated with the terminal nodes (i.e., the nodes labelled by terminal symbols) of *ott* for which no corresponding node exists in *itt*. We propose these attribute instances (normally set by the parser!) to be defined by *lexical evaluation rules* in terms of attribute instances of *itt*.

We denote a conditional tree transformation rule by *tr*: (*itt*, *ott*, *cond*, *eval*), where *itt* and *ott* are the input and the output tree template, *cond* is the enabling condition and *eval* is the set of lexical evaluation rules.

A conditional tree transformation rule *tr*: (*itt*, *ott*, *cond*, *eval*) is applicable to a subtree *IT* of an attributed derivation tree *T*1, if *itt* is an instance of *IT* and the evaluation of *cond* yields true. The application of *tr* consists of the creation of an instance *OT* of *ott* (in which the correspondence between subtrees and variables, established by *IT* and *itt*, is maintained) and the replacement of *IT* by *OT*. Moreover, it is assumed that the inner and the output attribute instances of *ott* are given the value *unknown*, and that the values of the synthesized attribute instances associated with the new terminal nodes of *ott* are computed using the rules specified by *eval*. Such an attributed derivation tree *T*2 may contain incorrect attribute values everywhere in *T*2, because of long-reaching attribute dependencies. A tree transformation may even cause the values of the input attribute instances of *ott* to be incorrect. To make the attribution of the tree correct again (which is generally needed in order to be able to test the predicates of subsequent tree transformations), a reevaluator should be activated, which works optimally in the number of visits to tree nodes and in the number of recomputations. By optimal we mean that, whenever possible, unnecessary reevaluations and superfluous visits to subtrees are avoided. The parallelization of such an incremental evaluator is outlined in the next section.

6. Parallel incremental attribute evaluation

The concurrent incremental evaluator we propose, is an incremental version of the combined static/dynamic evaluator of Böhm and Zwaenepoel [6]. The static part is an incremental version of the ordered attribute evaluation scheme [13]. The dynamic part is based on the reevaluation scheme of Reps, Teitelbaum and Demers [18]. For reasons of simplicity we assume that every tree transformation takes place in a region of the syntax tree which is processed either entirely statically or entirely dynamically. For example, we could let the reevaluation process start in a subtree which is evaluated entirely statically. The incremental static evaluator may compute a different value for a synthesized attribute instance of the root of the subtree, which is shared with the dynamic evaluator. This results in the activation of the incremental dynamic evaluator. From now on the cooperation of evaluation processes of the incremental evaluator proceeds in a similar way as in the non-incremental combined static/dynamic evaluator. As for the non-incremental evaluator, this may lead to a renewed activation of the static evaluator which started the reevaluation process.

6.1 Incremental static evaluation

In this section we sketch the incremental static evaluator, which is an incremental version of the ordered evaluator. Because the visit strategy for ordered attribute grammars [13] is used, the recomputation of each attribute instance is considered once, and only after its argument values have been reconsidered. Hence, the value of every attribute instance is guaranteed correct after the completion of the reevaluation process. The reevaluator works optimally in the sense that subtrees are skipped whenever possible, and the only attribute instances subject to reevaluation are those instances with an incorrect value, and those instances which directly depend on instances with incorrect values. Moreover, we restrict the reevaluation visits to the smallest possible subtree surrounding the restructured subtree (and the shortest path from the root of the tree to the root of this subtree). This ensures, in particular, that the reevaluator works in time linear in the size of the syntax tree and "almost" linear in the size of the "affected area" of the tree, i.e., all those nodes that have at least one wrong attribute value.

To mark the attribute instances that need to be reevaluated, we associate with every tree node a variable *NeedToBeEvaluated* of type *set of attributes*. To properly update *NeedToBeEvaluated*, we introduce a variable *Changed* of type *set of attributes*. In a downward visit of a node the variable *Changed* includes the inherited attribute instances of the node which have changed their value during the current visit. Similarly, in an upward visit *Changed* includes the synthesized attribute instances of the node whose values have changed during the current visit. Attribute a is inserted in *NeedToBeEvaluated* of N as soon as an argument of the attribute evaluation rule of a of N has changed, as indicated by *Changed*. Deletion is done immediately after the recomputation of a of N.

To improve the tree-walk strategy we associate with every tree node labelled by a nonterminal symbol a variable *SubtreeAffected* of type *set of visit numbers*. Let N_0 be a node, p: $X_{p0} \rightarrow X_{p1} \dots X_{pn_p}$ the production applied at N_0 and $N_1, \dots, N_{n_p}$ the sons of N_0 from left to right, respectively. Let $VS_p = VS_{p,1}, VS_{p,2}, \dots, VS_{p,m}$ be the visit-sequence (see [13]) of production p. In $VS_{p,i}$ a vertex labelled $v_{0,i}$ denotes the i-th downward visit of X_{p0} and a vertex labelled $v_{k,j}$ denotes the j-th upward visit of X_{pk}. *SubtreeAffected* of N_0 contains visit number i if and only if either $VS_{p,i}$ contains a defined attribute occurrence a of X_{pk}, such that $a \in$ *NeedToBeEvaluated* of N_k for some k $(0 \leq k \leq n_p)$, or $VS_{p,i}$ contains $v_{k,j}$, such that $j \in$ *SubtreeAffected* of N_k for some k $(1 \leq k \leq n_p)$. During reevaluation traversals *SubtreeAffected* of N_0 is updated when N_0 is exited during a visit.

This scheme guarantees a correct value for *SubtreeAffected* of N_0 whenever the reevaluator is not in the subtree with root N_0. This makes it possible to skip the subtree with root N_0 whenever the reevaluator returns to N_0 during a downward visit with number i, where $i \notin$ *SubtreeAffected* of N_0. At the end of the reevaluation process *SubtreeAffected* of N_0 is empty for all N_0.

Because of its recursive nature, the reevaluator starts at the root of the tree. So, at the moment of activating the reevaluator, *NeedToBeEvaluated* of N and *SubtreeAffected* of N are required to be correct for any node N in the syntax tree. To explain the initialization of these variables after a tree transformation we will consider both the original tree and the resulting tree. Immediately before the application of a tree transformation rule *tr*: (*itt*, *ott*, *cond*, *eval*) to a derivation tree *T*1, the values of all attribute instances of *T*1 are correct. Also, for every node *N* of *T*1, *NeedToBeEvaluated* of *N* and *SubtreeAffected* of *N* are empty. Since the values of the inner and output attribute instances are unknown, it follows that, as a consequence of a tree transformation, the following actions must be taken before the reevaluator is activated.

1. Every inner and every output attribute instance of *ott* must be included in its corresponding set *NeedToBeEvaluated*.
2. For every non-leaf node *N* of *ott* its associated variable *SubtreeAffected* of *N* must be set. From the fact that *SubtreeAffected* of *N* is defined in terms of variables associated with *N* itself and its immediate descendants, it follows that the instances of *SubtreeAffected* associated with the non-leaf nodes of *ott* can easily be computed from the bottom up.
3. For each ancestor of the root of *ott*, its associated instance *SubtreeAffected* has to be set. The initialization of these instances can be done during a bottom up tree-walk from the root of *ott* "straight" to the root of the tree.

A similar method is used by Engelfriet in [7], although he does not make use of sets *NeedToBeEvaluated* and *Changed*. A tree node is marked as affected for all future visits if one of its attribute instances changes its value. This may lead to unnecessary reevaluations and unnecessary visits to subtrees. Another (quite involved) solution is presented in [25]. An incremental evaluation algorithm for absolutely non-circular attribute grammars [14] can be found in [17].

The incremental evaluation method, sketched above, can easily be adjusted to pass-oriented attribute evaluation schemes, for which also variables *NeedToBeEvaluated* and *SubtreeAffected* are needed. The difference with incremental ordered attribute evaluation is that for simple multi-pass attribute evaluation [1, 5, 11] the variables *SubtreeAffected* are of type *set of pass numbers* (for more details see [2] and [3]).

6.2 Incremental dynamic evaluation

The incremental attribute evaluation method of Reps, Teitelbaum and Demers [18] is a natural extension of the dynamic evaluation scheme. Instead of a fixed dependency graph there is a continuously growing and shrinking dependency graph (called the "model") whose vertices represent possibly affected attribute instances that have not yet been reevaluated and whose arcs represent (direct or indirect) dependencies among these attribute instances. Every vertex of the model has either a label *R*, indicating that the associated attribute instance needs to be reevaluated, or a label *E* which means that its value may possibly remain equal (i.e., it is not yet known whether the associated attribute instance has to be reevaluated or not).

A vertex is removed from the model when the associated attribute instance is recomputed or when it becomes clear that its original value is correct. This implies that a vertex is ready for removal if it has in-degree 0. An attribute instance associated with such a vertex should be recomputed if it has label *R*, otherwise its original value is still correct. When the value of a recomputed attribute instance turns out to be changed, then every successor of this attribute instance should be labelled R.

If a tree transformation takes place in a subtree, then the starting model consists of the first affected attribute instance of the top tree. This attribute instance has label R. In case the tree transformation takes place in the top tree, then the starting model consists of the attribute instances of the output template *ott*, the direct dependencies among these attribute instances and the (indirect) dependencies induced by the context of *ott* in the actual syntax tree. Every vertex of the starting model associated with an input attribute instance of *ott* has label E and every vertex associated with an inner or an output attribute instance has label R.

At any moment the model corresponds to the currently affected area of the top tree. The model has to be expanded if an attribute instance a of a node N at the frontier of the model has changed its value and influences attribute instances in a neighbouring production p. The model is then extended as follows. First, the (indirect) dependencies at N, induced by the context of the old model, are removed. Second, the model is extended with the dependency graph of production p. Both graphs are pasted together along N. Third, new (indirect) dependencies, induced by the context of the new model, are added at the new frontier vertices. Finally, the vertices at the new frontier positions are labelled E, except the vertices associated with attribute instances depending on a. The expansion stops if no more attribute instances outside the model depend on changed attribute instances from the inside. The reevaluation process halts as soon as the model becomes empty.

From the fact that an attribute instance is only reevaluated after its in-degree has become 0, it follows that the recomputation of each attribute instance is considered once, and only when the reevaluation is guaranteed to yield its final value. Observe that the dynamic incremental evaluation algorithm works in time linear in the size of the affected area.

6.3 Concurrent incremental static/dynamic evaluation

The parallel incremental evaluator is a combination of the incremental static evaluator described in section 6.1, and the incremental dynamic evaluator described in section 6.2. The expansion of the model of the top tree may reach a statically evaluated subtree. This means that the model is extended with the attribute instances at the root of the subtree. When an inherited attribute instance of the root has both label R and in-degree 0, then the appropriate incremental static visit procedure is invoked. In a similar manner the affected area of a subtree may reach the dynamically evaluated top tree. When a synthesized attribute instance of the root of the subtree changes its value, then the model of the incremental dynamic evaluator is extended with the dependency graph of the production on top of the subtree, if it was not already a part of the model. New (indirect) dependencies are added at the new frontier positions, and the new vertices are labelled E, except the vertices depending on the attribute instance which just changed its value.

In the incremental combined static/dynamic evaluator each attribute instance is recomputed once, and only when the reevaluation is guaranteed to yield its correct value. The time complexity of the combined reevaluator is the same as that of the static evaluator. It works in time linear in the size of the tree and "almost" linear in the size of the affected area of the tree. In this, "almost" refers to the bottom trees, in which a tree walk repeatedly has to be made from the root to the affected area, and return.

7. Multiple asynchronous tree transformations

In [12] Kaplan and Kaiser presented a model for distributed program editing, which applies the dynamic attribute reevaluation scheme. In their work, a concurrent algorithm is first developed for incremental attribute evaluation in a single user, single edit environment. This algorithm is then expanded to handle multiple asynchronous edits, and then further extended to handle multiple asynchronous edits on program modules that are distributed across a number of workstations connected by a high speed network.

To be sure that the part of the tree a programmer is editing is not suddenly changed because of reevaluations propagated by another tree transformation, Kaplan and Kaiser introduce firewalls, which act as a barrier behind which a module can shelter if it is not ready to accept attribute change propagations from other modules. In our concurrent incremental static/dynamic evaluator, each region is either in its tree transformation phase or in its reevaluation phase. A subtree will only accept attribute change propagations from the top tree (and reversed) when it is in its reevaluation phase.

A significant difference between tree transformations for the purpose of program editing and conditional tree transformations is that the latter require the attribution of the tree to be made correct after every transformation. This is generally needed in order to be able to test the predicates of subsequent tree transformations. This implies, however, that conditional tree transformations cannot be performed in parallel. To remove this restriction, we formulate criteria which permit a delay in calling the reevaluator. These criteria allow in particular the execution of tree transformations in different regions at the same time, and also a delay in calling the reevaluator until a sequence of tree transformations has been performed and several parts of a region have been affected [4]. A delay in calling the reevaluator requires a different view of the correctness of attribute values of a syntax tree. For a non-circular attribute grammar, the classical theory defines one single value to be correct for each attribute instance. This value is called the *consistent* value of the attribute instance. For the purpose of conditional tree transformations we extend the classical attribute grammar framework by allowing a set of values to be correct for each attribute instance. Such a value is called *safe* [9, 10]. Every safe value should be an approximation of the consistent value, which is therefore the optimal safe value. The set of safe values contains the consistent value.

Tree transformations based on safe attribute values have the following characteristics.

1. If a tree transformation rule is applicable to a safely attributed tree, then it is also applicable to the corresponding consistently attributed tree.
2. A tree transformation applied to a safely attributed tree again yields a safely attributed tree.

These characteristics allow a tree transformation phase to perform without any interruption for reevaluations anywhere in the tree.

The safety of the conditional tree transformation rules is the responsibility of the writer of these rules, i.e., their safety is not checked at compiler generation time. However, in the next section we provide local criteria so that the writer can check the safety of his rules.

7.1 Safe trees and safe tree transformations

In the sequel we make a distinction between the local and global reevaluation of a transformed tree. The local reevaluation is restricted to the attribute instances in the restructured area of the tree (i.e., the area covered by *ott*). The global reevaluation is the incremental evaluation (according to the approach described in section 6.3) of the entire tree. We discuss a scheme where the reevaluation process after every tree transformation may be confined to the local reevaluation, and where the global reevaluation may be delayed.

Hereafter, we assume that for each attribute a the set $V(a)$ of possible values of a is partially ordered, and we denote this partial order by $\leq$ (in fact, this is ambiguous, because we should write $\leq_a$, but we want to keep our notation as simple as possible). For $x, y \in V(a)$, if $x \leq y$, then we say that x is an *approximation* of y, or that y is better ($\geq$) than x. For synthesized attributes of terminals we assume the partial order to be trivial, i.e., $x \leq y$ iff $x = y$. This is necessary because these attributes are imported attributes for which no evaluation rules are defined. For all other attributes we assume that the partial order has a smallest element, denoted (again ambiguously) by $\perp$. Informally, the value x of an attribute instance is called safe if $x \leq y$, where y is its consistent value.

For the comparison of safely and consistently attributed derivation trees, and for the expression of the requirements that guarantee the reliability of transformations based on safe syntax trees, we introduce the following notations and concepts.

Let T be an attributed syntax tree, then T^c denotes the result of the global reevaluation of T. More precisely, T^c is the unique consistently attributed tree with the same underlying syntax tree as T, and the same values for the corresponding synthesized attribute instances of the leaves.

For attributed trees T and T' with the same underlying syntax tree, $T \leq T'$ means that the value of every attribute instance of T is an approximation of the value of the corresponding attribute instance of T'. Note that if $T \leq T'$ then $T^c = T'^c$.

Now, the safety of (the values of the attribute instances of) a syntax tree, and the safety of a tree transformation rule, can be defined as follows.

Definition 1. T is *safe* iff $T \leq T^c$.

Note that T is consistent iff $T = T^c$; hence a consistent tree is safe.

Definition 2. A conditional tree transformation rule *tr* is safe if:

a) if *tr* is applicable to a subtree of a safely attributed tree, then *tr* is also applicable to that subtree of the corresponding consistently attributed tree,

b) the local reevaluation after the application of *tr* results in a safe tree.

Part a) of this definition says that a tree transformation rule is never wrongly applied. Part b) guarantees the reliability of subsequent transformations.

Using safety rather than consistency as the new definition of correctness we may conclude that during a tree walk, after the application of a tree transformation rule, the attribute instances may not have their best values, although their values are always safe. This means that during a walk where no global reevaluations are performed, every tree transformation is correct, although an interrupt of the walk in order to perform a global reevaluation (i.e., to compute the best values for all attribute instances) might have disclosed further opportunities for transformations during the continuation of the walk.

We now show that local restrictions can be imposed on the attribute evaluation and tree transformation rules that guarantee the safety of the tree transformation rules. First, we need the monotonicity of the evaluation rules and the enabling conditions. A function $f(x_1, x_2, ..., x_n)$ of attribute values, whose result is an attribute value, is *monotonic* if the following condition holds: if $a_i \leq b_i$ $(1 \leq i \leq n)$, and $f(a_1, a_2, ..., a_n)$, $f(b_1, b_2, ..., b_n)$ are defined, then $f(a_1, a_2, ..., a_n) \leq f(b_1, b_2, ..., b_n)$. An attribute evaluation rule or a lexical evaluation rule is *monotonic* if the function in its right part is monotonic. Note that the monotonicity of a lexical evaluation rule means that if $a_i \leq b_i$ then $f(a_1, a_2, ..., a_n) = f(b_1, b_2, ..., b_n)$. An enabling condition $f(x_1, x_2, ..., x_n)$ of a tree transformation rule is *monotonic* if the following condition holds: if $a_i \leq b_i$ $(1 \leq i \leq n)$ and $f(a_1, a_2, ..., a_n) =$ true, then $f(b_1, b_2, ..., b_n) =$ true (i.e., for false $\leq$ true f is monotonic).

Besides monotonic attribute evaluation rules, we also need for every tree transformation rule *tr*: (*itt*, *ott*, *cond*, *eval*) that "*ott* is *better* than *itt*". By this we mean that for every possible choice of values for the input attribute instances of *itt*, the values of the output attribute instances of *itt* are approximations of the values of the corresponding output attribute instances of *ott* (if they exist). Intuitively, this means that application of *tr* "increases the amount of information".

Definition 3. A tree transformation rule *tr*: (*itt*, *ott*, *cond*, *eval*) is *locally safe*, if:

a) *cond* is monotonic,

b) all lexical evaluation rules in *eval* are monotonic, and

c) *ott* is better than *itt*.

In [4] it has been proved that, for monotonic attribute evaluation rules, every locally safe tree transformation rule is safe. These criteria may help the writer of the attribute evaluation and tree transformation rules to check the safety of his rules. Another important conclusion from [4] is that, the use of both monotonic attribute evaluation rules and locally safe tree transformation rules yields an improvement of the attribute values at any local or global reevaluation.

7.2 Delayed incremental static evaluation

In section 5 we did not assume any local reevaluation was performed, i.e., all inner and output attribute instances of *ott* were given the value *unknown*. In section 7.1. we changed our strategy. This implies a change in the initialization of the variables *NeedToBeEvaluated* and *SubtreeAffected*. Observe that the local reevaluation of *ott* stops at the border of *ott*, i.e., all output attribute instances of *ott*. Let *a* **of** *K* be an output attribute instance of *ott* whose value has changed, then every attribute instance *b* **of** *N*, which depends on *a* **of** *K*, has to be inserted in *NeedToBeEvaluated* **of** *N*. The associated visit numbers should be included in *SubtreeAffected* of *N*'s ancestors in *ott* and in the production which borders at the root of *ott*.

We assume the tree transformations in a subtree to be performed during a tree walk which starts and finishes at the root. Hence, there is no need to make a separate bottom up tree walk from the root of *ott* to the root of the subtree in order to initialize the instances of *SubtreeAffected* on this path. These initializations can be done during the bottom up moves of the tree walk during which transformations are performed.

Observe that a single call of the incremental static evaluator of a subtree handles the reevaluations propagated from all the affected areas in the subtree, and also from attribute instances at the top of the subtree, which are affected by the incremental dynamic evaluator.

7.3 Delayed incremental dynamic evaluation

Every tree transformation in the top tree results in a graph which represents the corresponding affected area. Also propagations from reevaluations of subtrees may reach the top tree and result in affected areas. These graphs are merged when they overlap. This merging operation is a union operation, i.e., identical arcs and vertices in two graphs become one in the resultant graph.

The dependency graphs grow and shrink on-the-fly. As soon as an attribute instance has in-degree 0, it has certainly become independent of any reevaluations propagated from the output template associated with the tree transformation that ultimately caused its reevaluation. However, a graph connected with one transformation may grow to cover an attribute instance that has already been recomputed due to another transformation, and therefore, has already been released because of the shrinkage of the graph concerned. Kaplan and Kaiser [12] show that, for any tree on which *k* transformations take place asynchronously, in the worst case any attribute instance is reevaluated at most *k* times. In general, the number of reevaluations will be less because affected areas of different transformations overlap.

7.4 Concurrent delayed incremental combined static/dynamic evaluation

The system based on safe trees and safe tree transformations allows multiple asynchronous tree transformations and multiple asynchronous reevaluations. The only restriction is that every region (i.e., every subtree and also the top tree) is either in its transformation phase or in its reevaluation phase. In particular, this allows that one region is in its transformation phase while at the same time another region is in its reevaluation phase. A region will only accept attribute change propagations from another region when it is in its reevaluation phase.

8. Conclusion

We have presented the design of a concurrent incremental combined static/dynamic evaluator. The static part is an incremental version of the ordered attribute evaluation scheme. The dynamic part is an extension of the dynamic evaluation scheme.

To remove the restriction that every tree transformation should immediately be followed by an incremental reevaluation of the syntax tree, criteria have been formulated which permit a delay in calling the reevaluator. The use of safe trees and safe tree transformations allows multiple asynchronous tree transformations and multiple asynchronous reevaluations. The only restriction is that every region is either in its transformation phase or in its reevaluation phase. Different regions may be in different phases at the same time.

References:

1. Alblas, H.: A characterization of attribute evaluation in passes. *Acta Informatica* **16**, 427-464 (1981).
2. Alblas, H.: Incremental simple multi-pass attribute evaluation. *Proc. NGI-SION 1986 Symposium*, 319-342 (1986).
3. Alblas, H.: Optimal incremental simple multi-pass attribute evaluation. *Information Processing Letters* **32**, 289-295 (1989).
4. Alblas, H.: Iteration of transformation passes over attributed program trees. *Acta Informatica* **27**, 1-40 (1989).
5. Bochmann, G.V.: Semantic evaluation from left to right. *Comm. ACM* **19**, 55-62 (1976).
6. Böhm, H.-J. and Zwaenepoel, W.: Parallel attribute grammar evaluation. *Proc. 7th Int. Conf. on Distributed Computing Systems*, R. Popescu-Zeletin, G. Le Lam, K.H. Kim (eds), pp. 347-354. Berlin 1987.
7. Engelfriet, J.: Attribute grammars: Attribute evaluation methods. In: *Methods and tools for compiler construction*, pp. 103-138. Cambridge University Press 1984.
8. Frankel, J.: The architecture of closely-coupled distributed computers and their language processors. Ph.D. Thesis, Harvard University, 1983.
9. Ganzinger, H. and Giegerich, R.: A truly generative semantics directed compiler generator. *Proc. SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices* **17**, 6, 172-184 (1982).

10. Giegerich, R., Möncke, U. and Wilhelm, R.: Invariance of approximative semantics with respect to program transformations. *Informatik-Fachberichte* **50**, pp. 1-10. Springer 1981.
11. Jazayeri, M. and Walter, K.G.: Alternating semantic evaluator. *Proc. ACM 1975 Annual Conference*, 230-234 (1975).
12. Kaplan, S.M., Kaiser, G.E.: Incremental attribute evaluation in distributed language-based environments. *Proc. Fifth Annual ACM Symposium on the Principles of Distributed Computing*, 121-130 (1986).
13. Kastens, U.: Ordered attribute grammars. *Acta Informatica* **13**, 229-256 (1980).
14. Kennedy, K. and Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. *Proc. 3rd ACM Symposium on Principles of Programming Languages*, 32-49 (1976).
15. Kuiper, M.F.: Parallel attribute evaluation. Ph.D. Thesis, University of Utrecht, 1989.
16. Lipkie, D.E.: A compiler design for multiple independent processor computers. Ph.D. Thesis Dept. of Computer Science, University of Washington, Seattle, Wash., 1979.
17. Reps,T.W.: *Generating language-based environments*. The MIT Press (1983).
18. Reps, T., Teitelbaum, T. and Demers, A.: Incremental context-dependent analysis for language based editors. *ACM TOPLAS* **5**, 449-477 (1983).
19. Schell Jr., R.M.: Methods for constructing parallel compilers for use in a multi-processor environment. Ph.D. Thesis, Dept. of Computer Science, Report No. 958, University of Illinois at Urbana-Champaign, February 1979.
20. Seshadri, V.: Concurrent semantic analysis. CSRI 216, Computer Systems Research Institute, University of Toronto, September 1988.
21. Seshadri, V., Small, I.S. and Wortman, D.B.: Concurrent compilation. *Proc. IFIP WG 10.3 Working Conference on Distributed Processing*, M.H Barton, E.L Dagless, G.L. Reijns (eds), pp. 627-641. North-Holland 1988.
22. Seshadri, V., Wortman, D.B., Junkin, M.D., Weber, S., Yu, C.P. and Small I.S.: Semantic analysis in a concurrent compiler. *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* **23**, 7, 233-240 (1988).
23. Vandevoorde, M. T.: Parallel compilation on a tightly coupled multiprocessor. SRC Reports 26, Digital Systems Research Center, March 1988.
24. Wilhelm, R.: Computation and use of data flow information in optimizing compilers. *Acta Informatica* **12**, 209-225 (1979).
25. Yeh, D.: On incremental evaluation of ordered attribute grammars, *BIT* **23**, 308-320 (1983).