ATTRIBUTED TREE TRANSFORMATIONS WITH DELAYED AND SMART RE-EVALUATION

Henk Alblas

University of Twente, Department of Computer Science,
P.O. Box 217, 7500 AE Enschede, The Netherlands.

Abstract. Transformations of attributed program trees form an essential part of compiler optimizations. A tree transformation may invalidate attribute instances, not only in the restructured part of the tree but also elsewhere in the tree. To make the attribution of the tree correct again a re-evaluator has to be activated.

Criteria are formulated which allow a delay in calling the re-evaluator. These criteria allow a strategy of repeatedly applying alternate attribute evaluation and tree transformation phases. An attribute evaluation phase consists of a tree walk in which all attribute instances receive their correct values. A tree transformation phase consists of a tree walk in which as many tree transformations are performed as possible. The transformation phase is never interrupted to carry out a re-evaluation.

For re-evaluation purposes an incremental simple multi-pass evaluator is presented, which works optimally in the number of recomputations and in the number of visits to subtrees during each pass.

1. Introduction

In the classical theory [13] attribute grammars form an extension of the context-free grammar framework in the sense that information is associated with programming language constructs by attaching attributes to the grammar symbols representing these constructs. Attribute values are defined by attribute evaluation rules associated with the productions of the context-free grammar. These rules specify how to compute the values of certain attribute occurrences as a function of other attribute occurrences.

Compiler optimizations can be described by tree transformations, where complicated and non-efficient tree structures are replaced by equivalent but simpler and more efficient tree structures. For the specification of such tree transformations the classical attribute grammar framework has to be extended with conditional tree transformation rules [11, 15, 17], where predicates on attribute values (carrying context information) may enable the application of a transformation.

Traditionally, before the application of a tree transformation rule all attribute instances attached to the derivation tree are assumed to have correct values. A tree transformation may cause the values of some of the attribute instances within the derivation tree to become incorrect, which means that a renewed application of the attribute evaluation instructions will result in different values.

To make the attribution of a derivation tree correct again (which is generally needed in order to be able to test the predicates of subsequent tree transformations), a re-evaluation of the entire tree could be applied. However, a recomputation (after every application of a tree transformation rule) of all attribute instances attached to the tree is time consuming and should be avoided.

Instead, we formulate criteria which allow a delay in calling the re-evaluator until a sequence of tree transformations has been performed and several parts of the tree are assumed to be affected [4]. A delay in calling the re-evaluator requires a different view of the correctness of attribute values of a derivation 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. 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 should 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 re-evaluations 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, we do provide local criteria so that the writer can check the safety of his rules.

Practical examples show that, in general, after a sequence of transformations has been applied, continuation of the transformation phase is productive only after a complete re-evaluation of the entire derivation tree.

For such a "complete" re-evaluation we propose a "smart" simple multi-pass re-evaluator which keeps track of the attribute instances that need to be recomputed [2]. An attribute instance is marked as "needing recomputation" as soon as an argument of its attribute evaluation instruction has changed. Immediately after its recomputation this mark is removed. The tree traversal strategy can be improved by skipping unnecessary visits to subtrees. A subtree is "affected" for a certain pass if one of the attribute instances of one of its nodes needs to be recomputed during that pass. Otherwise the subtree can be skipped during that pass. For the pass-oriented approach this leads to an optimal incremental simple multi-pass evaluator that can be combined with any tree transformation strategy.

2. Basic Concepts

An *attribute grammar AG*, as defined in [13], is a context-free grammar G, which is augmented with attributes and attribute evaluation rules.

The underlying grammar G is a 4-tuple (V_N, V_T, P, S), where V_N and V_T denote the finite sets of nonterminal and terminal symbols, respectively, P is the finite set of productions and $S \in V_N$ is the start symbol, which does not appear in the right-hand side of any production. We write V for $V_N \cup V_T$.

Each symbol $X \in V$ has a finite set $A(X)$ of attributes, partitioned into two disjoint subsets $I(X)$ and $S(X)$ of *inherited* and *synthesized* attributes, respectively. For $X = S$ and $X \in V_T$ we require $I(X) = \varnothing$

The set of all attributes will be denoted by A, i.e., $A = \cup_{X \in V} A(X)$. Attributes of different grammar symbols are different. An attribute a of symbol X is also denoted by a **of** X. Each attribute a has a set $V(a)$ of possible values.

A production p is denoted as $X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$. Production p is said to have the *attribute occurrence* (a, p, k) if $a \in A(X_{pk})$.

The set of attribute occurrences of production p can be partitioned into two disjoint sets of defined occurrences and used occurrences denoted by $DO(p)$ and $UO(p)$, respectively.

These subsets are defined as follows:

$$DO(p) = \{(s, p, 0) \mid s \in S(X_{p0})\} \cup \{(i, p, k) \mid i \in I(X_{pk}) \wedge 1 \leq k \leq n\},$$

$$UO(p) = \{(i, p, 0) \mid i \in I(X_{p0})\} \cup \{(s, p, k) \mid s \in S(X_{pk}) \wedge 1 \leq k \leq n\}.$$

Associated with each production p is a set of *attribute evaluation rules* which specify how to compute the values of the attribute occurrences in $DO(p)$. The evaluation rule defining attribute occurrence (a, p, k) has the form

$$(a, p, k) := f((a_1, p, k_1), (a_2, p, k_2), ..., (a_m, p, k_m))$$

where $(a, p, k) \in DO(p)$, f is a total function and $(a_j, p, k_j) \in UO(p)$ for $1 \leq j \leq m$. We say that (a, p, k) *depends on* (a_j, p, k_j) for $1 \leq j \leq m$.

For each sentence of G a derivation tree exists. The nodes of the tree are labeled with symbols from V. For each non-leaf node there is a production $p: X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$, such that the node is labeled with X_{p0} and its sons with $X_{p1}, X_{p2}, ..., X_{pn}$, respectively. We say that p is the production *(applied) at* that node.

Given a derivation tree, instances of attributes are attached to the nodes in the following way: if node N is labeled with grammar symbol X, then for each attribute $a \in A(X)$ an instance of a is attached to node N. We say that the derivation tree has *attribute instance* a **of** N.

Let N_0 be a node, p the production at N_0, and $N_1, N_2, ..., N_n$ its sons from left to right, respectively. An *attribute evaluation instruction*

$$a \textbf{ of } N_k := f(a_1 \textbf{ of } N_{k1}, a_2 \textbf{ of } N_{k2}, ..., a_m \textbf{ of } N_{km})$$

is associated with attribute instance a **of** N_k if the attribute evaluation rule

$$(a, p, k) := f((a_1, p, k_1), (a_2, p, k_2), ..., (a_m, p, k_m))$$

is associated with production p. We say that a **of** N_k *depends on* a_i of N_{ki} for $1 \leq i \leq m$.

For each derivation tree T a *dependency graph* D_T can be defined by taking the attribute instances of T as its vertices. Arc (a **of** N_i, b **of** N_j) is contained in the graph if and only if attribute instance b **of** N_j depends on attribute instance a **of** N_i.

A path in a dependency graph will be called a *dependency path*, for which the following notation will be used: dp[a_1 **of** N_1, a_2 **of** N_2, ..., a_n **of** N_n] for $n > 1$ stands for a path composed of the arcs (a_1 **of** N_1, a_2 **of** N_2), (a_2 **of** N_2, a_3 **of** N_3), ..., (a_{n-1} **of** N_{n-1}, a_n **of** N_n).

An *attributed derivation tree* is a derivation tree where all attribute instances have a value (which is not necessarily consistent). A *consistently attributed derivation tree* is a derivation tree where the execution of any evaluation instruction does not change the values of the attribute instances.

The task of an attribute evaluator is to compute the values of all attribute instances attached to the derivation tree, by executing their associated evaluation instructions. Initially, the values of all attribute instances attached to the derivation tree are undefined, with the exception of the instances of the imported attributes. For simplicity we assume that the imported attributes are the synthesized attributes of the leaves of which the values are determined by the parser. The output of the evaluator is a consistently attributed tree.

3. Conditional Tree Transformations

We restrict ourselves to attributed tree transformations which preserve the syntax, i.e., all intermediate trees are derivation trees in the same context-free grammar.

For the definition of a tree template we need the concept of a "possibly" incomplete derivation tree where arbitrary symbols may label the root and the leaves. Normally, when we refer to a derivation tree, we mean a "complete" derivation tree, i.e., a derivation tree whose root is labeled with the start symbol and whose leaves are labeled with terminal symbols only. By a subtree we mean a subtree of a complete derivation tree.

To define conditional tree transformations, we first need to recall the definition of a purely syntactical tree transformation rule [6], consisting of two tree templates.

A *tree template* is a possibly incomplete derivation tree. Multiple occurrences of the same symbol as the label of a node are distinguished by subscripts. So, in general, node labels are of the form X_i, where X is a terminal or nonterminal and i a subscript. Nonterminal symbols (possibly with a subscript) labeling the leaves are the *variables* of the tree template.

An *instance* of a tree template is created by substituting for each variable of the tree template a subtree whose root has the same nonterminal as the variable.

A *tree transformation rule* is a pair (*itt*, *ott*) of tree templates, such that all variables occurring in *ott* also occur as variables in *itt*; *itt* and *ott* are called the *input tree template* and the *output tree template*, respectively.

A tree transformation rule (*itt*, *ott*) is *applicable* to a subtree *IT* of a derivation tree *T*1, if *itt matches* the top of *IT*, i.e., if *IT* is an instance of *itt*. The fact that *IT* is an instance of *itt* establishes a relation between the variables of *itt* and subtrees of *IT*.

The application of tree transformation rule (*itt*, *ott*) consists of the creation of an instance *OT* of *ott* in which the relation between subtrees of *OT* and variables of *ott* is the same as established by matching *itt* with *IT*. The resulting subtree *OT* replaces subtree *IT* of *T*1, thus creating a new derivation tree *T*2. Note that by the definition of tree templates (the variables of *ott* must be different) duplication of a subtree of *IT* in *OT* is excluded.

The fact that *itt* and *ott* are (possibly incomplete) derivation trees in the same grammar guarantees that for each application of a tree transformation rule (*itt*, *ott*) the instance *IT* of *itt* and the corresponding instance *OT* of *ott* are in the same grammar.

To guarantee that *OT* correctly fits in the surrounding tree *T*2 (i.e., that the production applied above *OT* preserves the syntax), it is necessary and sufficient to require that grammar symbol *A* labeling the root of *itt* may be replaced by grammar symbol *B* labeling the root of *ott*, at any occurrence of *A* in the right part of any production. However, for reasons of simplicity we impose an additional requirement on the transformation rules, namely that *itt* and *ott* have equally labeled roots (which also have the same index). Observe that it is always possible to extend *itt* and *ott* with an extra production so that their roots are labeled equally. Similarly, we require both the input template and the output template to consist of more than one node.

Syntactically (i.e., for attribute-free derivation trees), the applicability of a tree transformation rule to a subtree is confined by the above-mentioned matching criterion. It may be further restricted by contextual information, collected and distributed by attributes. For this we need to associate attributes with tree templates.

Let X_i be the label of a node of a tree template *tt*, where X is a grammar symbol and i denotes its subscript in *tt*. The subscript may be omitted in the case of a single occurrence of X in *tt*. We say that tree template *tt* has *attribute instance* (a, tt, X_i) if $a \in A(X)$. (a, tt, X_i) is an *inherited* instance if $a \in I(X)$, and a *synthesized* instance if $a \in S(X)$.

Let (*itt*, *ott*) be a tree transformation rule. Attribute instances in *itt* and *ott* are said to *correspond* if they are the same attribute of equally labeled nodes, i.e., they are of the form (*a*, *itt*, *Y*) and (*a*, *ott*, *Y*). This notion is only relevant for attribute instances of the root and the leaves of *itt* and *ott*. Note that every attribute instance in *ott* of the root or the variables has a corresponding attribute instance in *itt*.

Having associated attributes with tree templates in a natural way, the transformation rules can be extended by *enabling conditions* [11, 15, 17] which are predicates on attribute instances of the input template.

Next, we focus on the attribution of a derivation tree after the application of a tree transformation rule. The difference between the original tree and the restructured tree is effected by the replacement of the input template by the output template. No syntactical changes take place in the subtrees substituted for corresponding variables of *itt* and *ott*. Notice also that the production above the restructured subtree remains unchanged, since we required *itt* and *ott* to have equally labeled roots. So, we assume that the attribute instances of the subtrees substituted for the variables of *itt* and *ott* keep their values after a transformation. The same holds for the attribute instances of the tree part above *itt* and *ott*. Thus, to obtain a fully (but perhaps not correctly) attributed derivation tree we could restrict ourselves to the evaluation of the attribute instances of *ott*.

The set of attribute instances of a tree template can naturally be partitioned into three disjoint subsets of input, output and inner attribute instances.

Definition 3.1. With respect to a tree template, the *input attribute instances* are the inherited attribute instances of its root and the synthesized attribute instances of its leaves; the *output attribute instances* are the synthesized attribute instances of its root and the inherited attribute instances of its leaves; the *inner attribute instances* are the attribute instances of the inner nodes. □

Observe that the inner and the 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, however, needed for the synthesized attribute instances associated with the terminal nodes of *ott* for which no corresponding node exists in *itt*. We propose these attribute instances (normally set by the parser!) to be defined, as part of the tree transformation rule, by *lexical evaluation rules* in terms of attribute instances of *itt*.

The synthesized attribute instances of terminal symbols of *ott*, for which a corresponding terminal symbol exists in *itt*, are assumed to be copied from *itt*.

Having informally introduced the concept of a conditional tree transformation we are now ready to give the following definition.

Definition 3.2. A conditional tree transformation rule is a 4-tuple *tr*: (*itt*, *ott*, *cond*, *eval*), where

- *itt* and *ott* are the *input* and the *output tree template*, respectively. *itt* and *ott* must have equally labeled roots (with the same subscript) and must contain more than one node. All variables occurring in *ott* also occur as variables in *itt*.
- *cond* is the *enabling condition*, a predicate on attribute instances of *itt*.
- *eval* is the set of *lexical evaluation rules* which specify the computation of the synthesized attribute instances of the new terminal nodes of *ott* in terms of attribute instances of *itt* (in the case *cond* yields true). □

A conditional tree transformation rule *tr*: (*itt*, *ott*, *cond*, *eval*) is *applicable* to a subtree *IT* of a derivation tree *T*, if *itt* matches the top of *IT* and the evaluation of *cond* yields true.

The *application* of transformation rule *tr* consists of the steps (1), (2), and (3), and possibly (4):

(1) Creation of an instance *OT* of *ott* (in which the correspondence between subtrees and variables, established by *IT*, is maintained) and the replacement of *IT* by *OT*, thus creating a (partially attributed) derivation tree *T*2.

(2) Computation of the values of the synthesized attribute instances associated with the new terminal nodes of *ott*, using the rules specified by *eval*.

(3) The local re-evaluation phase: evaluation of the attribute instances in the restructured area of *T*2 (i.e., the area covered by *ott*).

(4) The global re-evaluation phase: re-evaluation of all attribute instances of *T*2 (except, of course, the synthesized attribute instances of the leaves).

The *full application* of *tr* consists of (1), (2), (3), and (4) and the *(partial) application* of *tr* consists of (1), (2), and (3). Note that both types of application result in a (completely) attributed derivation tree. The full application results in a consistently attributed derivation tree, whereas the attributed derivation tree resulting from the (partial) application of *tr* may contain inconsistencies.

Conditional tree transformation rule *tr*: (*itt*, *ott*, *cond*, *eval*) will be written as follows:

tr: **transform** *itt* **cond** *cond* **into** *ott* **eval** *eval* **end.**

It is allowed to leave out the part "**cond** *cond*" if *cond* is true and the part "**eval** *eval*" if *eval* is empty.

We illustrate the application of transformations with an example, taken from [17], which concerns constant folding. For the specification of tree templates we use the following linear notation for trees: within angular brackets the root is followed by its sequence of subtrees, comma symbols act as separators and we write *a* **of** *Y* for the attribute instance (*a*, *tt*, *Y*) of a tree template *tt*.

Observe that the notation *a* **of** *Y* for attribute instance (*a*, *itt*, *Y*) and (*a*, *ott*, *Y*) leads to the same notation for corresponding attribute instances in *itt* and *ott*.

Example 3.1. The conditional tree transformation rule

```
tr: transform <expr,ident>
        cond element (idno of ident) in: (pool of expr)
          into <expr,const>
            eval val of const :=
                 value-of (idno of ident) in: (pool of expr)
    end
```

describes the replacement of an identifier by a constant, as depicted in Figure 1. Terminal symbol *ident* has a synthesized attribute *idno* indicating its number, assigned by the parser. Terminal symbol *const* has a synthesized attribute *val* representing its value. Inherited attribute *pool* **of** *expr* contains a pair (*idno*, *val*) for each variable whose value is known to have the same value whenever control passes through this point. Here, *idno* represents the identifier number of the variable and *val* its associated value.

The transformation of an identifier into a constant is enabled if its identifier number is found in the pool of constant variables. Its associated value will be assigned to synthesized attribute *val* of the newly created constant.

The function "element (*idno*: integer) in: (*p*: pool) **delivers** boolean:" checks whether a pair with first component *idno* is found in *p* or not. The function "value-of (*idno*: integer) in: (*p:* pool) **delivers** integer:" returns the value of *idno* in pool *p*. □

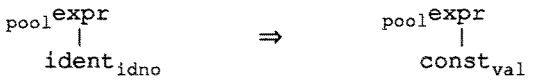

Fig.1. Replacement of a variable by a constant.

4. Delay of re-evaluation

In Section 3 we explained that the result of the partial application of a tree transformation rule on a consistently attributed derivation tree $T1$ will be a fully attributed tree $T2$, which may, however, contain inconsistencies. This is caused by the fact that, in general, the values of some of the output attribute instances of *ott* in $T2$ will differ from the values of their corresponding output attribute instances of itt in $T1$. Let a **of** $N1$ be an output attribute instance of *ott* whose new value differs from its old value. Then in $T2$, every attribute instance b **of** $N2$, such that the dependency graph D_{T2} includes a dependency path dp[a **of** N_1, ..., b **of** N_2], may have a wrong value. A tree transformation may even cause the values of the input attribute instances of *ott* to be incorrect (and hence the inner and the output instances as well).

Hence, if a correct value is required for every attribute instance in $T2$, then the local re-evaluation phase has to be followed by a global re-evaluation phase, unless for every output attribute instance of *ott* in $T2$ its value is equal to the value of its corresponding output attribute instance of *itt* in $T1$.

We now discuss a strategy where the re-evaluation process after each tree transformation may be confined to the local re-evaluation phase, and where the global re-evaluation phase may be delayed. Thus, in the following we always assume the partial application of a tree transformation.

The classical theory on attribute grammars defines one single value to be correct for each attribute instance of any derivation tree (of which the values of the synthesized attribute instances of the leaves are given). For our tree transformation strategy, where re-evaluations may be restricted to *ott*, we extend the classical attribute grammar framework by allowing a set of values to be correct for each attribute instance. Each value of such a set should be an approximation of the correct value according to the classical attribute grammar definition.

In [9, 10] the new correct values are called *safe*, whereas the old correct values are called *consistent*. We also use this terminology.

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 $x \leq y$, 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 $\perp$. As an example, $V(a)$ may be the set of all finite sets of identifiers, ordered by set-inclusion, with the empty set as the smallest element.

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 derivation trees, we introduce the following notations and concepts.

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

For attributed derivation trees $T1$ and $T2$, subtree IT of $T1$ and tree transformation rule tr: $T1\ [IT] \text{—}tr\rightarrow T2$ means that tr is applicable to IT of $T1$, with $T2$ the result of the (partial) application. Note that $T2^c$ is the result of the full application.

The purpose of a set C of conditional tree transformation rules, for a given consistently attributed derivation tree T, is to produce another consistently attributed derivation tree T' such that T' is obtained from T by a sequence of full applications of rules of C. This is formalized as follows. T' is consistently derivable from T by C if

either $T' = T$

or there is a subtree IT of T, a rule $tr \in C$, and an attributed derivation tree $T1$ such that $T\ [IT] \text{—}tr\rightarrow T1$ and T' is consistently derivable from $T1^c$ by C.

Of course, one would normally continue applying the rules of C until no rule of C is applicable anymore.

Note that if T' is consistently derivable from T then this can always be realized by a number of tree traversals, during which the rules are applied in their proper order.

We now want to define a condition on the transformation rules so that their partial application can be used rather than their full application. The idea is to use approximations of the consistently derivable trees rather than the consistently derivable trees themselves.

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 $T^c = T'^c$.

We are now ready to formally define the safety of (the values of the attribute instances of) a derivation tree, and the safety of a tree transformation rule.

Definition 4.1. T is *safe* iff $T \leq T^c$. □

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

Definition 4.2. A conditional tree transformation rule tr is safe if:
If $T1\ [IT] \text{—}tr\rightarrow T2$, and $T1$ is safe, then
a) $T1^c\ [IT] \text{—}tr\rightarrow T2'$, and
b) $T2 \leq T2'^c$,
for some $T2'$. □

Part a) of this definition says that 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. Part b) says that the result of the partial application is an approximation of the result of the full application. Note that it is also a safe approximation. In fact, from part b) we know that $T2 \leq T2'^c$, and from this it follows that $T2^c = T2'^c$, and so, $T2 \leq T2^c$. Thus we obtain the following fact: a safe transformation rule preserves safety of trees; this 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 re-evaluations are performed, every tree transformation is correct, although an interrupt of the walk in order to perform a global re-evaluation (i.e., to compute the best values for all attribute instances) might have disclosed further opportunities for transformations during the continuation of the walk.

A tree $T2$ is safely derived from a consistently attributed input tree $T1$ if $T2$ is the result of a sequence of safe tree transformations applied to $T1$. It can easily be shown from Definition 4.2 that by a global re-evaluation of the safely derived tree $T2$ an output tree $T2^c$ is obtained which is consistently derivable from the input tree $T1$. This allows an attribute evaluation and tree transformation algorithm where the evaluation phase alternates with the transformation phase. This process is repeated until no more tree transformations are possible.

We now want to 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:

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:

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 4.3. 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,
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.

Practical examples show that, in general, after a traversal of an entire derivation tree, during which tree transformations are performed, a subsequent traversal is productive only after a global re-evaluation of the tree.

5. A smart re-evaluator

Throughout this paper we assume that the attribute evaluation strategy is *simple multi-pass*, which means that a fixed number of depth-first left-to-right traversals (called passes) are made over the derivation tree and all instances of the same attribute are evaluated during the same pass. From [1] we repeat some terminology and definitions concerning simple multi-pass evaluation.

A partition of the set of attributes A into a sequence of mutually disjoint subsets will be denoted by $<A_0, A_1, ..., A_m>$, where A_0 includes all synthesized attributes of terminal symbols (whose values should be computed by the parser before the evaluator is started).

A partition $<A_0, A_1, ..., A_m>$ of the set of attributes A is *correct* if A_0 consists of the synthesized attributes of the terminal symbols and the instances of all attributes in set A_i $(1 \le i \le m)$ can be evaluated during the i-th pass of the simple multi-pass evaluator.

An attribute grammar is *simple m-pass* if a correct partition $<A_0, A_1, ..., A_m>$ of the set of attributes A exists. An attribute grammar is *simple multi-pass* if it is simple m-pass for some m.

For each partition $<A_0, A_1, ..., A_m>$ of the set of attributes A of an attribute grammar a *pass function* pass: $A \rightarrow \{0, 1, ..., m\}$ can be defined as pass$(a) = i$ if $a \in A_i$. The pass function is *correct* if the partition is correct.

The following tree-walk algorithm defines a simple multi-pass evaluator. Each call of "visit subtree (*root*, *i*)" corresponds to a pass. Node N is denoted by either "$N.0$" or "N" and "k-th son of N" is written as "$N.k$".

Algorithm 5.1. Simple multi-pass evaluation.
Input: an attributed derivation tree where only the synthesized attribute instances of the terminal symbols are defined;
a correct partition $<A_0, A_1, ..., A_m>$ of the set of attributes A.
Output: an attributed derivation tree where all attribute instances are defined.
Algorithm:

```
begin
    const m = ...;
    type node = ...;
         pass number = 1..m;
    var root: node;
    procedure visit subtree (N: node; i: pass number);
    begin {let p: X_p0 → X_p1 X_p2 ... X_pn be the production at node N}
        for k from 1 to n
        do for all a ∈ I(X_pk)
           do if a ∈ A_i
              then evaluate a of N.k
              fi
           od;
           if X_pk ∈ V_N
           then visit subtree (N.k, i)
           fi
        od;
        for all a ∈ S(X_p0)
        do if a ∈ A_i
           then evaluate a of N.0
           fi
        od
    end {of visit subtree};
    read (root);
    for i from 1 to m  {i: pass number}
    do visit subtree (root, i) od
end  □
```

Algorithm 5.1 describes the (re-)evaluation of all attribute instances attached to a derivation tree. Below we present a more efficient re-evaluator from [2], which does no unnecessary calculations and skips subtrees when possible.

To be able to mark the attribute instances that need to be evaluated, we associate with every tree node a variable *NeedToBeEvaluated* of type *set of attributes*. For *NeedToBeEvaluated* associated with node N we use the same notation as for attributes, namely *NeedToBeEvaluated* **of** N.

To update *NeedToBeEvaluated* properly, we introduce a global variable *Changed* of type *set of attributes*. During a pass, a node will be visited downwards and upwards. In a downward visit of a node the variable *Changed* includes those inherited attribute instances which have changed their value. Similarly, in an upward visit *Changed* includes those synthesized attribute instances whose values have changed during the current pass.

Attribute a is inserted in *NeedToBeEvaluated* **of** N as soon as an argument of the evaluation instruction of a **of** N has changed, as indicated by variable *Changed*. Deletion is done immediately after the recomputation of a **of** N.

We now try to improve the tree-walk strategy. A subtree is visited during the i-th pass if *Changed* is not empty and/or the instance of *NeedToBeEvaluated* attached to the root or one of its descendants includes attributes with pass number i.

We associate with every tree node labeled by a nonterminal symbol a variable *SubtreeAffected* of type *set of pass numbers*. Let N_0 be a node, p: $X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$ the production applied at N_0 and $N_1, N_2, ..., N_n$ the sons of N_0 from left to right, respectively. *SubtreeAffected* **of** N_0 includes pass number i if and only if

either a defined attribute occurrence (a, p, k) exists, such that pass(a **of** X_{pk}) = i and $a \in$ *NeedToBeEvaluated* **of** N_k, for some k $(0 \leq k \leq n)$.

or $i \in$ *SubtreeAffected* **of** N_k, for some k $(1 \leq k \leq n)$.

During re-evaluation passes, *SubtreeAffected* **of** N_0 will be updated at the following times:

1. When N_0 is visited, pass number i will be inserted in *SubtreeAffected* **of** N_0 as soon as one of the requirements specified above is found to be fulfilled.
2. Pass number i will be deleted from *SubtreeAffected* **of** N_0 when N_0 is visited for the second time during the i-th pass.

This guarantees a correct value for *SubtreeAffected* **of** N_0 whenever N_0 is visited during the re-evaluation process, provided that, when visiting a node, the instances of *NeedToBeEvaluated* and *SubtreeAffected* associated with all the nodes of its subtree (i.e., rooted in this node) are correct. At the end of the re-evaluation process *SubtreeAffected* **of** N_0 is empty for all N_0.

The re-evaluator starts at the root of the derivation tree. The moment the re-evaluator is activated, *NeedToBeEvaluated* **of** N and *SubtreeAffected* **of** N are required to be correct for any node N in the derivation tree. To explain the initialization of these variables we consider the effect of the partial application of a tree transformation.

The result of the partial application of a tree transformation rule tr = (*itt*, *ott*, *cond*, *eval*) to a fully and safely attributed derivation tree $T1$, is a fully and safely attributed tree $T2$. Observe that the local re-evaluation of attribute instances of $T2$ stops at the border of *ott*, i.e., at the output attribute instances of *ott*. Let a **of** K be an output attribute instance of *ott* whose new value differs from its old value. This means that every attribute instance b **of** N such that D_{T2} contains an arc (a **of** K, b **of** N) has to be inserted in *NeedToBeEvaluated* **of** N. This holds for the attribute instances of the productions which "border on *ott*", i.e., the production applied immediately above the root of *ott* and the productions applied at the leaves of *ott*.

First, we treat the productions which border on a leaf of *ott*. Let N_0 be a node, p: $X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$ the production applied at N_0 and $N_1, N_2, ..., N_n$ the sons of N_0 from left to right, respectively. Let N_0 be a leaf of *ott* and a **of** N_0 an output attribute instance of *ott* whose value has changed, then every attribute instance b **of** N_k $(0 \leq k \leq n)$ such that (b, p, k) depends on $(a, p, 0)$ has to be inserted in *NeedToBeEvaluated* **of** N_k, and the associated pass number pass(b **of** N_k) has to be inserted in *SubtreeAffected* **of** N_0.

The variable *SubtreeAffected* of every non-leaf node N of *ott* must include all the pass numbers contained in the variables *SubtreeAffected* of all the leaves of *ott*. The values of these variables can be computed from the bottom up.

Next, we treat the production which borders on the root of *ott*. Let N_0 be a node, $p: X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$ the production applied at N_0 and $N_1, N_2, ..., N_n$ the sons of N_0 from left to right, respectively. Let N_j $(1 \leq j \leq n)$ be the root of *ott* and a **of** N_j an output attribute instance of *ott* whose value has changed, then every attribute instance b **of** N_k $(0 \leq k \leq n)$ such that (b, p, k) depends on (a, p, j) has to be inserted in *NeedToBeEvaluated* **of** N_k, and the associated pass number pass(b **of** N_k) has to be inserted in *SubtreeAffected* **of** N_0.

All the pass numbers contained in *SubtreeAffected* of the root of *ott* (i.e., N_j) must be inserted in *SubtreeAffected* **of** N_0. Every ancestor of N_0 must include the same pass numbers as *SubtreeAffected* **of** N_0. The initialization of these instances of *SubtreeAffected* will be done during the bottom up moves of the tree walk to the root of the derivation tree.

Simple multi-pass re-evaluation using this scheme is defined in Algorithm 5.2. In this algorithm we use the statement "re-evaluate a **of** N", by which we mean the following steps:

```
old := a of N;
evaluate a of N;
new := a of N;
if old ≠ new then insert a in Changed fi;
delete a from NeedToBeEvaluated of N.
```

Procedure "visit subtree" in Algorithm 5.2 is an adapted version of the one presented in Algorithm 5.1. Procedure "propagate change" (with parameters k of type integer and p of type production number) inserts attribute instances, for which one of the arguments is found in *Changed*, in their corresponding sets *NeedToBeEvaluated* and updates the associated set *SubtreeAffected* accordingly. For $k = 0$, *Changed* contains inherited attribute instances associated with the left-hand side of a production, and for $k \neq 0$, *Changed* contains synthesized attribute instances associated with the k-th symbol of the right-hand side.

Algorithm 5.2. Simple multi-pass re-evaluation.

Input: an attributed derivation tree where all attribute instances have a safe value but some attribute instances may have an inconsistent value;
sets *NeedToBeEvaluated* and *SubtreeAffected* associated with tree nodes;
a correct partition $<A_0, A_1, ..., A_m>$ of the set of attributes A.

Output: an attributed derivation tree where all attribute instances have consistent values.

Algorithm:

```
begin
    const m = ...;
          EmptySetOfAttributes = [];
    type node = ...;
         production number = 1..r;
         pass number = 1..m;
         attributes = (... enumeration of attributes ...);
         set of attributes = set of attributes;
    var root: node;
        Changed: set of attributes;
```

```
procedure visit subtree (N: node; i: pass number);
begin {let p: X_p0 → X_p1 X_p2 ... X_pn be the production at node N}
    procedure propagate change (k: integer;
                                p: production number);
    begin {p: X_p0 → X_p1 X_p2 ... X_pn is the production with number p}
       if Changed ≠ EmptySetOfAttributes
       then for j from 0 to n
            do for all a ∈ A(X_pj)
               do for all b ∈ A(X_pk)
                  do if (a,p,j) depends on (b,p,k))
                        and b ∈ Changed
                     then insert a in NeedToBeEvaluated of N.j;
                          insert pass(a of X_pj) in
                                          SubtreeAffected of N
                     fi
                  od
               od
            od
       fi
    end {of propagate change};
    {insertion of attributes in NeedToBeEvaluated of N.j (0≤j≤n),
     and pass numbers in SubtreeAffected of N}
    propagate change (0, p);
    for k from 1 to n
    do Changed := EmptySetOfAttributes;
       {recomputation of inherited attribute instances of N.k;
        deletion of these attribute instances
        from NeedToBeEvaluated of N.k;
        insertion of attribute instances in Changed}
       for all a ∈ I(X_pk)
       do if a ∈ A_i and a ∈ NeedToBeEvaluated of N.k
          then re-evaluate a of N.k
          fi
       od;
       if X_pk ∈ V_N
       then if Changed ≠ EmptySetOfAttributes
               or i ∈ SubtreeAffected of N.k
            then visit subtree (N.k, i);
                 {insertion of attributes in
                  NeedToBeEvaluated of N.j (0≤j≤n),
                  and pass numbers in SubtreeAffected of N}
                 propagate change (k, p)
            fi
       fi
    od;
    Changed := EmptySetOfAttributes;
    {recomputation of synthesized attribute instances of N.0;
     deletion of these attribute instances
     from NeedToBeEvaluated of N.0;
     insertion of attribute instances in Changed}
    for all a ∈ S(X_p0)
    do if a ∈ A_i and a ∈ NeedToBeEvaluated of N.0
       then re-evaluate a of N.0
       fi
    od;
    delete i from SubtreeAffected of N;
    for j from 1 to n
    do if X_pj ∈ V_N
       then SubtreeAffected of N :=
            SubtreeAffected of N ∪ SubtreeAffected of N.j
       fi
    od
end {of visit subtree};
```

```
    read(root);
    for i from 1 to m
    do if i ∈ SubtreeAffected of root
       then Changed := EmptySetOfAttributes;
            visit subtree (root, i)
       fi
    od
end □
```

In [5] the approach of Algorithm 5.2 is adjusted to the visit sequences of ordered attribute grammars [12].

Algorithm 5.2 works optimally in the number of re-evaluations of attribute instances and the number of visits to subtrees. The space required for bookkeeping information (one bit for every attribute or pass number in every instance of *NeedToBeEvaluated* and *SubtreeAffected*) is linear in the size of the tree.

Observe that Algorithm 5.2 works in time linear in the size of the tree, but not in the size of its affected areas. Because of the recursive nature of Algorithm 5.2 every pass starts at the root of the tree and walks down to the affected areas. In [3], for the case where the re-evaluator is called after every tree transformation, an iterative version of Algorithm 5.2 is presented which works linear in the size of the affected area of the tree. This version of the re-evaluator starts its first pass at the root of the restructured subtree. This subtree is called the "subtree under consideration". If, at the end of a pass over the subtree under consideration, the value of *Changed* turns out to be non-empty, then the current subtree under consideration is widened, i.e., the father of its root becomes the root of the new subtree under consideration. The pass is then continued over the widened tree. This process of widening continues until the value of *Changed* is empty. Every subsequent pass starts at the root of the subtree under consideration.

A method similar to the iterative version is used by Engelfriet in [7] for l-ordered attribute grammars [8], although he does not make use of sets *NeedToBeEvaluated* and *Changed*. A tree node is marked to be affected for all coming visits after one of its attribute instances gets a different value. This may lead to unnecessary re-evaluations and superfluous visits to subtrees. Another solution for l-ordered attribute grammars is presented in [18].

Solutions for arbitrary non-circular attribute grammars are discussed in [14, 16]. Möncke et all. in [14] discuss the re-evaluation of attributes which become incorrect as a result of an optimizing tree transformation. Reps et all. in [16] give a solution for the updating of attribute values during an editing session.

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 (1986), pp. 319-342.
3. Alblas, H.: Optimal incremental simple multi-pass attribute evaluation. Memorandum INF 86-27, University of Twente (1986).
4. Alblas, H.: Iteration of transformation passes over attributed program trees. Memorandum INF 87-28, University of Twente (1987).
5. Alblas, H.: Attribute evaluation methods. Memorandum, University of Twente, to appear (1988).
6. DeRemer, F.L.: Transformational grammars. In: Compiler Construction: An advanced course, F.L. Bauer, J. Eickel (eds.). Lecture Notes in Computer Science, Vol. 21. Springer 1974, pp. 121-145.
7. Engelfriet, J.: Attribute grammars: Attribute evaluation methods. In: Methods and tools for compiler construction. Cambridge University Press 1984, pp. 103-138.
8. Engelfriet, J., Filè, G.: Simple multi-visit attribute grammars, JCSS 24, 283-314 (1982).

9. Ganzinger, H. and Giegerich, R.: A truly generative semantics directed compiler generator. In: Proc. SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17, 6 (1982), pp. 172-184.
10. Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximative semantics with respect to program transformations. Informatik-Fachberichte 50, Springer 1981, pp. 1-10.
11. Glasner, I., Möncke, U., Wilhelm, R.: OPTRAN, a language for the specification of program transformations. Informatik-Fachberichte 34, Springer 1980, pp. 125-142.
12. Kastens, U.: Ordered attribute grammars. Acta Informatica 13, 229-256 (1980).
13. Knuth, D.E.: Semantics of context-free languages. Math. Systems Theory 2, 127-145 (1968). Correction in: Math. Systems Theory 5, 95-96 (1971).
14. Möncke, U., Weisgerber, B., Wilhelm, R.: How to implement a system for manipulation of attributed trees. Informatik-Fachberichte 77, Springer 1984, pp. 112-127.
15. Nestor, J.R., Mishra, B., Scherlis, W.L., Wulf, W.A.: Extensions to attribute grammars. Technical Report TL 83-36, Tartan Laboratories Inc., 1983.
16. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language based editors. ACM TOPLAS 5, 449-477 (1983).
17. Wilhelm, R.: Computation and use of data flow information in optimizing compilers. Acta Informatica 12, 209-225 (1979).
18. Yeh, D.: On incremental evaluation of ordered attribute grammars, BIT 23, 308-320 (1983).