Acta Informatica 27, 1–40 (1989)

Acta Informatica

Iteration of Transformation Passes over Attributed Program Trees

Henk Alblas

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

Contents page

Summary. Transformations of attributed program trees form an essential part of compiler optimizations. A strategy of repeatedly applying alternate attribute evaluation and tree transformation phases is discussed. An attribute evaluation phase consists of a sequence of passes over the tree. A tree transformation phase consists of a single pass, which is never interrupted to carry out a re-evaluation. Both phases can be performed in parallel. This strategy requires a distinction between consistent (i.e., correct) and approximate attribute values. Tree transformations can be considered safe if they guarantee that the attribute values everywhere in the program tree will remain consistent or will become at least approximations of the consistent values, so that subsequent transformations can be applied correctly.

This attribute evaluation and tree transformation strategy shows similarities with the evaluation methods for circular attribute grammars.

1. Introduction

Attribute grammars have proved to be a useful formalism for specifying the syntax and the static semantics of programming languages, as well as for implementing editors, compilers, translator writing systems and compiler generators.

Several methods have been developed to evaluate the semantic attributes within the derivation tree of a program. An overview is given in [7].

In this paper we restrict ourselves to the simple left-to-right multi-pass evaluation strategy [1, 4], where 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.

Conditional tree transformations form an essential part of compiler optimizations. For the specification of such transformations the classical attribute grammar framework has to be extended with attributed tree transformation rules [11, 14, 16], where predicates on attribute values may enable the application of a transformation. Such a conditional tree transformation rule includes: an input template (describing the structure of the tree part to which the transformation has to be applied), an output template (describing the structure of the transformed part of the tree), enabling conditions which are predicates on attribute instances of the input template, and, possibly, rules which define the values of the attribute instances that are normally available before the evaluation process starts, i.e., the synthesized attribute instances associated with the terminal symbols of the output template.

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, a re-evaluation of the entire tree could be applied. However, a repeated computation of all the attribute instances after every transformation is inefficient and should be avoided. Several methods have been developed to minimize the number of recomputations and the number of visits to subtrees [2, 7, 13, 15, 17]. These methods have in common that they assume a re-evaluation of the affected attributes of the tree to be performed after every tree transformation.

In this paper we consider a different approach in the sense that the re-evaluation process will be delayed until a sequence of tree transformations has been performed and the entire tree is expected to be affected. This approach requires a different view of the correctness of attribute values in a derivation tree. For a non-circular attribute grammar, the classical theory defines one single value to be correct for each attribute instance. This is also 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. More precisely, for each attribute there is a partial order $\leqq$ on its possible values, and a value x of an attribute instance is safe iff $x \leqq y$, where y is its consistent value. Thus, the consistent value is the optimal safe value.

In this paper we study tree transformations which preserve the safety of the attribute values in the derivation tree. Our research was stimulated by the ideas stated in [9, 10].

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 some local criteria so that the writer can check the safety of his rules.

The use of safe tree transformation rules allows a tree transformation and re-evaluation strategy with the following characteristics.

1) Tree transformations are performed during a pass over the derivation tree.

2) The re-evaluation of attribute instances in the derivation tree is delayed until a transformation pass has been finished. (Note that the attribute instances of the area corresponding to the output template receive a value as part of the tree transformation).

3) The attribute evaluation phase (which consists of a fixed number of passes) and the tree transformation phase (which consists of one pass) are performed alternately, until it turns out that no more tree transformations are possible.

4) The attribute evaluation phase and the tree transformation phase may also be combined, if required.

The method of the alternate (or combined) application of attribute evaluation and tree transformation phases, presented in this paper, shows similarities with the evaluation methods for circular attribute grammars, presented by Babich and Jazayeri in [3] and Farrow in [8]. Each method, in its own way, improves the attribute values by repeatedly traversing the derivation tree.

This paper is organized as follows: Section 2 provides an introduction to the classical theory of attribute grammars and summarizes the principles of simple left-to-right multi-pass evaluation. Conditional tree transformations are defined in Sect. 3. In Sect. 4 the safety criteria for conditional tree transformation rules are developed, which allow the delay of a re-evaluation phase, which concerns the entire derivation tree, after every tree transformation. In Sect. 5 the alternate or combined application of an attribute evaluation and a tree transformation phase is compared to the evaluation method for circular attribute grammars. Both methods are applied to an example which concerns constant folding, constant propagation and dead code elimination in Sect. 6. Concluding remarks are made in Sect. 7.

2. Basic Concepts

An *attribute grammar* AG, as defined in [12], is a context-free grammar augmented with attributes and attribute evaluation rules. The underlying grammar G is a 4-tuple (V_N, V_T, P, S). The finite sets V_N of nonterminal and V_T of terminal symbols form the vocabulary $V = V_N \cup V_T$. P is the set of productions and $S \in V_N$ is the start symbol, which does not appear in the right part of any production. The grammar G is assumed to be reduced in the sense that every nonterminal symbol is accessible from the start symbol and can generate a string of terminal symbols only.

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. The start symbol should not have inherited attributes.

The set of all attributes will be denoted by A, i.e., $A = \bigcup_{X \in V} A(X)$. Attributes of different grammar symbols are considered as different. If necessary we will denote an attribute a of symbol X by a **of** X. With each attribute a a set $V(a)$ of possible values is associated.

Let P consist of r productions, numbered from 1 to r and let the p-th production be

$$X_{p0} \to X_{p1} X_{p2} \ldots X_{pn}$$

where $n \geqq 0$, $X_{p0} \in V_N$ and $X_{pk} \in V$ for $1 \leqq k \leqq n$.

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 will be denoted by $AO(p)$. This set 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:

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

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

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

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

For each sentence of G a derivation tree exists. For the definition of a tree transformation rule we also need the concept of a "possibly incomplete" derivation tree where arbitrary symbols may label the root and the leaves. Apart from that, by 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.

The nodes of a (possibly incomplete) derivation tree are labeled with symbols from V. For each inner node a production p: $X_{p0} \to X_{p1} X_{p2} \ldots X_{pn}$ exists, such that the node is labeled with X_{p0} and its sons with $X_{p1}, X_{p2}, \ldots, X_{pn}$, 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 $N_1, N_2, \ldots, N_n$ its sons from left to right, respectively. An *attribute evaluation instruction*

$$a \textbf{ of } N_k := f(a_1 \textbf{ of } N_{k_1}, a_2 \textbf{ of } N_{k_2}, \ldots, a_m \textbf{ of } N_{k_m})$$

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

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

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

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.

If D_T is acyclic, its arcs specify a partial ordering of the attribute instances. The existence of arc (a **of** N_i, b **of** N_j) indicates that attribute instance a **of** N_i must be computed before attribute instance b **of** N_j.

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** $N_2, \ldots, a_n$ **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). A path $dp[a_1$ **of** $N_1, \ldots, a_n$ **of** N_n, a_1 **of** $N_1]$ will be called a *circular dependency path*. An attribute grammar is *circular* if it includes a derivation tree whose dependency graph contains a circular dependency path, otherwise the attribute grammar is non-circular. Unless stated otherwise, we assume an attribute grammar to be non-circular.

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. In general the order of evaluation is free, with the only restriction that an attribute evaluation instruction cannot be executed before the values of its arguments are available. 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 derivation tree.

In this paper the attribute instances are evaluated during a bounded number of passes over the derivation tree, where a pass is a depth-first left-to-right traversal of the tree. Note that (for the sake of simplicity) we do not allow right-to-left traversals. We further restrict the evaluation strategy to be *simple multi-pass* [1, 4], which means that with each attribute a fixed pass number can be associated so that the evaluation of all its instances in any derivation tree of the grammar can be performed in that pass.

We assume the reader to be familiar with attribute evaluation in passes. 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 $\langle A_0, A_1, \ldots, A_m \rangle$, 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 $\langle A_0, A_1, \ldots, A_m \rangle$ 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 $A_i (1 \leqq i \leqq m)$ 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 $\langle A_0, A_1, \ldots, A_m \rangle$ 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 $\langle A_0, A_1, \ldots, A_m \rangle$ of the set of attributes A of an attribute grammar a *pass function* pass: $A \to \{0, 1, \ldots, m\}$ can be defined as pass$(a) = i$ if $a \in A_i$. The pass function is *correct* if the partition is correct.

3. Conditional Tree Transformations

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

To define conditional tree transformations we first recall the definition of a purely syntactical tree transformation rule [6], composed 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 indices. So, in general, node labels are of the form $X[i]$, where X is a terminal or nonterminal and i an index. Nonterminal symbols (possibly with an index) 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, (and if the roots of itt and ott are labeled by the same nonterminal, then these nonterminals should have the same index); 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 $T1$, if

1) itt *matches* the top of IT, i.e., IT is an instance of itt.

2) ott *fits in* the surrounding tree, i.e., if $A[i]$ and $B[j]$ label the roots of itt and ott, respectively, and $X \to \alpha A \beta \in P$ is the production applied immediately above IT in $T1$, then also $X \to \alpha B \beta$ must be in P (or $A = B = S$).

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 $T1$, thus creating a new derivation tree $T2$. 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.

Syntactically (i.e., for attribute-free derivation trees), the applicability of a tree transformation rule to a subtree is confined by the above-mentioned criteria.

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 index in tt. The index may be omitted in the case of a single occurrence of X in tt. We say that tree template tt has *attribute instance* $(a, \text{tt}, X[i])$ if $a \in A(X)$. $(a, \text{tt}, X[i])$ 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 *corresponding* if they are the same attribute of identically 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.

Having associated attributes with tree templates in a natural way, the transformation rules can be extended by *enabling conditions* [11, 14, 16] 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 and, in the event of differently labeled template-roots, by a change of the production applied immediately above the restructured subtree. No syntactical changes take place elsewhere in the tree (except for the case that an entire subtree is deleted).

From the fact that the attribute evaluation rules are associated with the productions it follows that after every application of a tree transformation rule the attribute evaluator can be re-activated in order to execute (at least) the attribute evaluation instructions associated with the newly included productions, i.e., the productions of the output template and possibly the production immediately above the restructured subtree. However, special actions have to be taken for the synthesized attribute instances associated with the new terminal nodes of the output template (new in the sense that the label of such a node does not occur in itt). We propose these attribute instances (normally set by the parser!) to be defined, as part of the tree transformation rule, by *lecixal evaluation rules* in terms of attribute instances of the input template.

Let (itt, ott) be a tree transformation rule, and let (a, ott, Y) be an attribute instance, associated with a new terminal symbol Y of ott. A lexical evaluation rule for (a, ott, Y) has the form

$$(a, \text{ott}, Y) := f((a_1, \text{itt}, Y_1), (a_2, \text{itt}, Y_2), \ldots, (a_m, \text{itt}, Y_m))$$

where f is a partial function and (a_j, itt, Y_j) is an attribute instance of itt, for $1 \leq j \leq m$.

The synthesized attribute instances of terminal symbols of the output template, for which a corresponding terminal symbol exists in the input template, are assumed to be copied from the input template.

We assume tree transformations to be performed during a sequence of left-to-right transformation passes over the derivation tree (possibly interrupted by re-activations of the attribute evaluator) and distinguish two possibilities to apply a tree transformation during such a pass.

Consider a subtree which may be restructured by the application of a tree transformation rule. During a pass over the tree the root of the subtree will be visited twice: the first time during a *downward move* and the second time during an *upward move.* Visiting the root for the first time the transformation could be applied when entering the subtree (i.e., before visiting the descendants of the root). Visiting the root for the second time the transformation could be done when leaving the subtree. So, for each tree transformation rule we will specify when it has to be applied, either during a downward move or during an upward move.

For our pass-oriented approach we therefore use the following definition of a conditional tree transformation rule.

Definition 3.1. A *conditional tree transformation rule* is a 5-tuple tr: (dir, itt, ott, cond, eval), where

- dir is the *direction* of the move at the moment when the transformation has to be tried. The domain of dir is $\{up, down\}$.
- itt and ott are the *input* and the *output tree template*, respectively. All variables occurring in ott also occur as variables in itt. If the roots of itt and ott are labeled by the same nonterminal, then these nonterminals have the same index.
- 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). □

Note that the lexical evaluation rules are only used in case the predicate cond is true. Thus, if $(a_1, \text{itt}, Y_1), \ldots, (a_n, \text{itt}, Y_n)$ are all attribute instances of itt, cond is the predicate

$$p((a_{i_1}, \text{itt}, Y_{i_1}), \ldots, (a_{i_k}, \text{itt}, Y_{i_k}))$$

and the lexical evaluation rule

$$(a, \text{ott}, Y) := f((a_{j_1}, \text{itt}, Y_{j_1}), \ldots, (a_{j_m}, \text{itt}, Y_{j_m}))$$

is in eval, then we require in Definition 3.1 that for all $x_1, \ldots, x_n$ with $x_i \in V(a_i)$: if $p(x_{i_1}, \ldots, x_{i_k}) = \text{true}$ then $f(x_{j_1}, \ldots, x_{j_m})$ is defined.

A conditional tree transformation rule tr: (dir, itt, ott, cond, eval) is *applicable* to a subtree IT, if the following conditions are satisfied:

- itt matches the top of IT;
- ott fits in the surrounding tree;
- the evaluation of cond yields true.

Making a pass over an attributed derivation tree a tree transformation rule tr: (dir, itt, ott, cond, eval) can be applied to a subtree IT, after a downward or an upward move to the root of IT, if tr is applicable to IT and, if in addition the direction of the move corresponds with the value of dir.

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 $T2$.

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

(3) Evaluation of the attribute instances in the restructured area of $T2$ (i.e., the area covered by ott and, in the case of differently labeled template roots, the production applied immediately above OT);

(4) Re-evaluation of all attribute instances of $T2$ (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. Application of tr will be described in more detail in Sect. 4.

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

tr: **transform** dir 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.

Conditional tree transformation rules with the same input template and the same direction may be combined as follows:

tr: **transform** dir itt **cond** cond_1 **into** ott_1 **eval** eval_1
⋮
cond cond_n **into** ott_n **eval** eval_n
end.

We illustrate the application of tree transformations with two small examples, taken from a more comprehensive example which concerns data flow analysis, in particular constant folding, constant propagation and dead code elimination, in Sect. 6.

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. We write *a* **of** *Y* for the attribute instance (a, tt, Y) of a tree template tt. To simplify our notation we allow indices of different occurrences of the same grammar symbol in itt and ott to be deleted if there is no need to distinguish these grammar symbols in cond and eval.

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

trans1: **transform up** ⟨whilestat, **while,** ⟨cond, boolconst⟩, **do,** stats, **od**⟩
cond boolval **of** boolconst = true
into ⟨loop-forever, **forever, do,** stats, **od**⟩
cond boolval **of** boolconst = false
into ⟨no-operation⟩
end

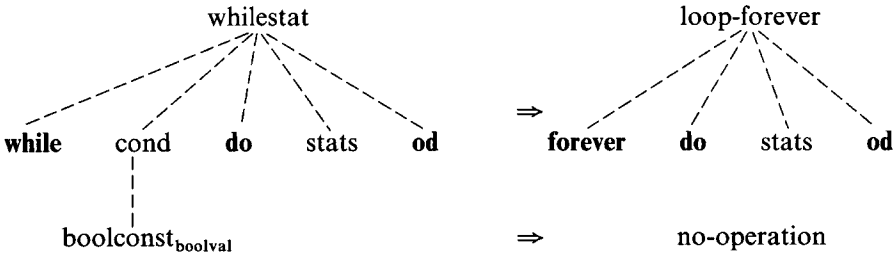

Fig. 1 Replacement of a while statement by a loop-forever or a no-operation

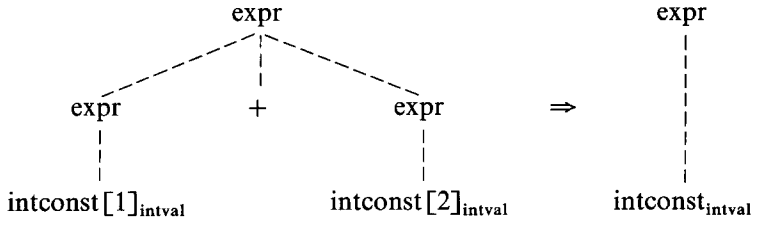

Fig. 2. Compile-time evaluation of a constant expression

describes the replacement of a while statement by a loop-forever or a no-operation, as illustrated in Fig. 1, in the form itt⇒ott. Note that stats is the only variable of itt (boolconst is a terminal). □

Example 3.2. The unconditional tree transformation rule

```
trans2: transform up ⟨expr, ⟨expr, intconst[1]⟩, +, ⟨expr, intconst[2]⟩⟩
        into ⟨expr, intconst⟩
          eval intval of intconst := intval of intconst[1] +
                                     intval of intconst[2]
        end
```

describes the compile-time evaluation of constant expressions. The input template in Fig. 2 shows two instances of synthesized attribute intval of terminal symbol intconst. The sum of these values is assigned to the instance of intval in the output template. □

4. Iteration of Evaluation and Tree Transformation Phases

Steps (1) and (2) of the application of a tree transformation rule tr: (dir, itt, ott, cond, eval) to an attributed derivation tree $T1$ result in a partially attributed derivation tree $T2$. To make the attribution of $T2$ complete again step (3), and possibly also step (4), could be performed. We repeat the purpose of both steps:

(3) The local re-evaluation phase, restricted to the restructured part of $T2$ (i.e., the area associated with ott and, if necessary, the production applied immediately above ott).

(4) The global re-evaluation phase, for the whole of $T2$.

To describe the local re-evaluation phase in detail, we extend the occurrences of the input and the output template in $T1$ and $T2$, if necessary, such that the resulting templates have identically labeled roots and consist of more than one node. Identically rooted versions of the input and the output template, both consisting of more than one node, whether an extension took place or not, will be called *complete* input and output templates and will be denoted by compl-itt and compl-ott, respectively.

No extension is needed if itt and ott already have roots with the same label and already consist of more than one node, i.e., in this case compl-itt = itt and compl-ott = ott.

Otherwise, itt and ott need to be extended with an extra production as follows: Let production $X_{p0} \to X_{p1} \dots X_{pk} \dots X_{pn}$ be applied immediately above itt in $T1$, with X_{pk} labeling the root of itt and let production $X_{q0} \to X_{q1} \dots X_{qk} \dots X_{qn}$ be applied immediately above ott in $T2$, with X_{qk} labeling the root of ott. Clearly $X_{pi} = X_{qi}$ for $0 \leqq i \leqq n, i \neq k$. The extensions compl-itt and compl-ott of itt and ott, respectively, are constructed as follows: Consider an incomplete derivation tree, composed of a node labeled X_{p0} and n sons labeled $X_{p1}, \dots, X_{pk}, \dots, X_{pn}$, respectively. Now, replace the node labeled X_{pk} by itt to form compl-itt. Observe that the leaves of compl-itt are both new leaves labeled with grammar symbols $X_{pi} (1 \leqq i \leqq n, i \neq k)$ from the right part of production p and old leaves from itt. A similar approach is followed to construct compl-ott from production q and ott.

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

Definition 4.1. For a complete 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. □

We now come back to the attribution of the restructured area of the derivation tree. To start with, it is assumed that in step (1) the attribute instances of the subtrees substituted for the variables of compl-itt and compl-ott have kept their values after the transformation. The same holds for the attribute instances of the tree part surrounding compl-itt and compl-ott (including their roots).

Moreover, the evaluation of attribute instances of compl-ott is preceded by step (2), i.e., by the computation of the synthesized attribute instances of the new terminal nodes of ott, as specified by eval. Also, the synthesized attribute instances of the identically labeled terminal nodes of itt and ott are assumed to keep their values. The same holds for the synthesized attribute instances of the terminal nodes of the productions above itt and ott. So, when starting

the local re-evaluation process, all the input attribute instances of compl-ott have a value. This implies that the attribute evaluator, suitably adapted, is able to compute the inner and output attribute instances of compl-ott, using the ordinary attribute evaluation instructions.

In general, the values of some of the output attribute instances of compl-ott in $T2$ will differ from the values of their corresponding output attribute instances of compl-itt in $T1$. Let a **of** N_1 be an output attribute instance of compl-ott whose new value differs from its old value. Then, in $T2$ every attribute instance b **of** N_2, such that the dependency graph D_{T2} includes a dependency path $dp[a \textbf{ of } N_1, \ldots, b \textbf{ of } N_2]$, may have an incorrect value. A tree transformation may even cause the values of the input attribute instances of compl-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 the derivation tree, then the local re-evaluation phase has to be followed by a global re-evaluation phase, unless for every output attribute instance of compl-ott in $T2$ its value is equal to the value of its corresponding output attribute instance of compl-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 the restructured area, 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 [5, 8–10, 14].

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

Assumption 4.1. Hereafter, we asume that for each attribute a the set $V(a)$ of possible values of a is partially ordered, and we denote this partial order by $\leqq$ (in fact, this is ambiguous, because we should write $\leqq_a$, but we want to keep our notation as simple as possible). For $x, y \in V(a)$, if $x \leqq y$, we say that x is an *approximation* of y, or that y is *better* ($\geqq$) than x. For synthesized attributes of terminals we assume the partial order to be trivial, i.e., $x \leqq y$ 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 $x \leqq y$, 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 trans-

formations based on safe derivation trees, we introduce the following notations and concepts.

Notation. 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. □

Notation. For attributed derivation trees $T1$ and $T2$, subtree IT of $T1$ and tree transformation rule tr, $T1\,[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T2$ 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 $\mathrm{tr} \in C$, and an attributed derivation tree $T1$ such that $T[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T1$ 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 transformation passes, during which the rules are applied in their proper direction.

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 those trees themselves.

Notation. For attributed trees T and T' with the same underlying syntax tree, $T \leqq T'$ means that the value of every attribute instance of T is an approximation (in the sense of Assumption 4.1) of the value of the corresponding attribute instance of T'. Note that if $T \leqq T'$ then $T^c = T'^c$ (using the triviality of the partial order of the values of synthesized attributes of terminals). □

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 4.2. T is *safe* iff $T \leqq T^c$. □

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

Definition 4.3. A conditional tree transformation rule tr is *safe* if:

If $T1\,[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T2$, and $T1$ is safe,
then a) $T1^c\,[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T2'$, and
b) $T2 \leqq T2'^c$,
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 first (partial) application is an approximation of the result of the second (full) application. Note that it is also a safe approximation. In fact, from part b) we know that $T2 \leqq T2'^c$, and from this it follows that $T2^c = T2'^c$, and so, $T2 \leqq T2^c$. 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 pass over a derivation tree, after the application of a tree transformation rule and during the continuation of the pass, the attribute instances may not have their best values, although their values are always safe. This means that during a pass where no global re-evaluations are performed, every tree transformation is correct, although an interrupt of the pass in order to make extra tree traversals for re-evaluation purposes (i.e., to compute the best values for all attribute instances) might have disclosed further opportunities for transformations during the continuation of the pass [9, 10].

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 simply be shown from Definition 4.3 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 leads to the following evaluation and transformation algorithm for a simple m-pass attribute grammar. First, m evaluation passes are made to compute the consistent value for every attribute instance in the derivation tree. Second, a tree transformation pass is made in which as many tree transformations are applied as possible. This process of making a sequence of evaluation passes followed by a single transformation pass is repeated until no more tree transformations are possible.

Algorithm 4.1. Attribute evaluation and conditional tree transformations for a simple m-pass attribute grammar with partially ordered attribute domains, and a set of safe conditional tree transformation rules.

Input: A derivation tree T of which only the values of the synthesized attribute instances of the leaves are available (more formally an attributed derivation tree T of which the values of all attribute instances are $\perp$, except the values of the synthesized attribute instances of the leaves).

Output: An attributed derivation tree, consistently derivable from T^c, to which no conditional tree transformation rule is applicable.

Algorithm:

repeat
for i **from** 1 **to** m
do perform the i-th evaluation pass **od**;
perform a transformation pass during which as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

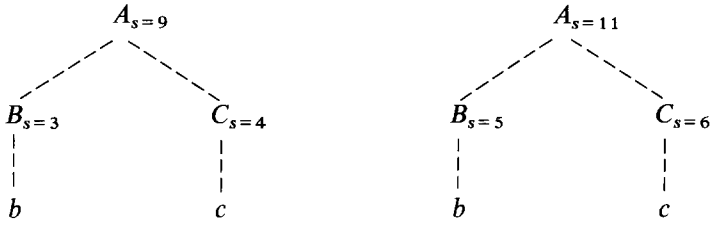

Fig. 3. Derivation trees with safe and consistent attribute values

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. For this we need the monotonicity of the evaluation rules and the enabling conditions.

A function $f(x_1, x_2, \ldots, x_n)$ of attribute values, whose result is an attribute value, is *monotonic* if:

if $a_i \leqq b_i (1 \leqq i \leqq n)$, and
$f(a_1, a_2, \ldots, a_n), f(b_1, b_2, \ldots, b_n)$ are defined,
then $f(a_1, a_2, \ldots, a_n) \leqq f(b_1, b_2, \ldots, b_n)$.

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 $a_i \leqq b_i$ then $f(a_1, a_2, \ldots, a_n) = f(b_1, b_2, \ldots, b_n)$ (if they exist).

An enabling condition $f(x_1, x_2, \ldots, x_n)$ of a tree transformation rule is *monotonic* if:

if $a_i \leqq b_i (1 \leqq i \leqq n)$ and $f(a_1, a_2, \ldots, a_n) = \text{true}$,
then $f(b_1, b_2, \ldots, b_n) = \text{true}$.
(i.e., for false $\leqq$ true f is monotonic).

Statement 4.1. In the following we restrict ourselves to attribute grammars whose attribute evaluation rules are monotonic. □

Note that, in general, the execution of monotonic attribute evaluation rules preserves the safety of trees, but does not necessarily improve their attribute values. Indeed, the attribute values may even become worse. This is shown in the following (unrealistic) example.

Example 4.1. Figure 3 shows two attributed versions of the same derivation tree. The attribute values are non-negative integers with the usual ordering and 0 as the bottom element. The attribute instances and their values are shown in the trees. Let the (monotonic) evaluation rule s **of** $A := s$ **of** $B + s$ **of** C be associated with production $A \to BC$, and let the rules s **of** $B := 5$ and s **of** $C := 6$ be associated with productions $B \to b$ and $C \to c$, respectively. Thus, the left tree is safely attributed (because $T \leqq T^c$) and the right tree is consistently attributed (and is, in fact, T^c). Application of the evaluation rule for s **of** A in the context of the left tree delivers the value 7 for s **of** A which is still safe, but not an improvement, compared to the current safe value. □

Consider a tree transformation rule tr: (dir, itt, ott, cond, eval). What happens locally to compl-itt and compl-ott during application of tr to a consistent tree, is determined completely by the values of the input attribute instances of compl-itt (by consistency, all attribute instances of compl-itt are determined by the attribute evaluation rules; the attribute instances of compl-ott are determined by the lexical evaluation rules, the attribute evaluation rules, and the attribute instances taken over from compl-itt, as indicated by steps (2) and (3) of the application of tr). We say that compl-ott is *better* than compl-itt if for every possible choice of values for the input attribute instances of compl-itt, the values of the output attribute instances of compl-itt are approximations of the values of the corresponding output attribute instances of compl-ott (if they exist). Intuitively, this means that application of tr "increases the amount of information".

Definition 4.4. A tree transformation rule tr: (dir, itt, ott, cond, eval) is *locally safe*, if:

(a) cond is monotonic,
(b) all lexical evaluation rules in eval are monotonic,
(c) for every possible compl-itt and compl-ott (extensions of itt and ott) compl-ott is better than compl-itt. □

Definition 4.5. An attributed tree T is *locally safe* if, for every attribute evaluation instruction of T, its execution leads to a better ($\geqq$) value for the attribute instance that is computed. □

This means that attribute evaluation improves the tree. Two facts are important.

(1) The execution of one attribute evaluation instruction to a locally safe tree leads again to a locally safe tree (by monotonicity of the attribute evaluation rules).

(2) If a tree is locally safe, then it is safe.

(*Proof of (2):* Call the attribute evaluator for a locally safe tree T. By Definition 4.5 and by (1), the attributed trees obtained after each step of the evaluator form an ascending chain. The output of the evaluator is T^e. Hence $T \leqq T^e$.)

The following theorem states a local criterion for the safety of a tree transformation rule.

Theorem 4.1. *A locally safe conditional tree transformation rule* tr *is safe.*

Proof. Requirement a) of Definition 4.3 is implied by condition (a) of Definition 4.4 and the safety of $T1$.

To prove requirement b) of Definition 4.3, consider the transformations

$T1\,[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T2$, where $T1$ is safe, and

$T1^e\,[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T2'$.

The monotonicity of the attribute evaluation rules, condition (b) of Definition 4.4, and the safety of $T1$ imply: $T2 \leqq T2'$.

The next step to prove is: $T2' \leqq T2'^c$. Condition (c) of Definition 4.4 says that, in the application $T1^c[\mathrm{IT}] \xrightarrow{\mathrm{tr}} T2'$, every output attribute instance of compl-ott has a better value than the corresponding output attribute instance of compl-itt. Observe however, that the smaller values from compl-itt have been used as arguments in attribute evaluation instructions for $T1^c$. Also observe, that the values of all attribute instances in the context of compl-ott in $T2'$ have been copied from $T1^c$. So, by monotonicity, the execution of an attribute evaluation instruction leads to a better value for any attribute instance that depends directly on output attribute instances of compl-ott, and (since $T1^c$ is consistent) to the same value for any other attribute instance. This means that $T2'$ is locally safe and hence, by (2) above, safe.

Finally, $T2 \leqq T2'$ and $T2' \leqq T2'^c$ imply (by transitivity of $\leqq$) that $T2 \leqq T2'^c$. □

In the proof above we have shown that application of a locally safe tree transformation rule to a consistent tree yields a locally safe tree. Similarly, we can prove the important fact:

(3) Application of a locally safe tree transformation rule to a locally safe tree yields again a locally safe tree.

We now investigate whether it is useful to perform attribute evaluations during the transformation pass.

Recall that monotonic attribute evaluation rules and safe tree transformation rules preserve the safety of trees, but do not always guarantee better values for the attribute instances that are recomputed. However, by (1) and (3) above, the use of both monotonic attribute evaluation rules and locally safe tree transformation rules preserves the local safety of trees and thus yields an improvement at any execution of an attribute evaluation instruction.

Intuitively, such an improvement is desirable because it may lead to the earlier applicability of transformation rules. (We have not pursued this formally; informally we will assume in what follows that the improvement of attribute values has a positive effect on tree transformation algorithms). Thus, for a simple m-pass attribute grammar with monotonic attribute evaluation rules, the following kind of combination of attribute evaluations and locally safe tree transformations is attractive.

Algorithm 4.2. Attribute evaluation and conditional tree transformations for a simple m-pass attribute grammar with partially ordered attribute domains and monotonic attribute evaluation rules, and a set of locally safe conditional tree transformation rules.

Input: A derivation tree T of which only the values of the synthesized attribute instances of the leaves are available (more formally an attributed derivation tree T of which the values of all attribute instances are $\perp$, except the values of the synthesized attribute instances of the leaves).

Output: An attributed derivation tree, consistently derivable from T^c, to which no conditional tree transformation rule is applicable.

Algorithm:
repeat
for *i* **from** 1 **to** *m*
do perform the *i*-th evaluation pass **od**;
perform a pass during which **some** attribute instances are evaluated, and as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

One could also think of a complete mixture of attribute evaluations and tree transformations, as follows (note that the tree in which all attribute instances, except the synthesized attribute instances of the leaves, have the value $\perp$ is locally safe).

Algorithm 4.3.
Initialize all attribute instances of the derivation tree with $\perp$, except the synthesized attribute instances of the leaves;
repeat
perform a pass during which *all* attribute instances are evaluated, and as many tree transformations are applied as possible
until no tree transformations were applied during the last $m+1$ passes. □

Practical examples show that, in general, a subsequent transformation pass is productive only after a complete re-evaluation of the entire derivation tree. From this it follows that preference should be given to Algorithm 4.1, or to Algorithm 4.2 on the condition that the attributes to be computed during the transformation pass are selected carefully.

In the following algorithms we keep the requirements that the attribute evaluation rules are monotonic and the tree transformation rules are locally safe.

In Algorithms 4.1 and 4.2 every transformation pass (except the last one) is preceded by the m-th pass of the previous evaluation phase and followed by the first pass of the next evaluation phase. In the remainder of this section we will investigate whether Algorithm 4.2 can be sped-up by moving the evaluations of the attributes with pass number 1 and pass number m to the tree transformation pass.

We first adapt Algorithm 4.2 such that the attributes with pass number m are both computed during the m-th evaluation pass and during the transformation pass. This is expressed by Algorithm 4.4.

Algorithm 4.4.
repeat
for *i* **from** 1 **to** *m*
do perform the *i*-th evaluation pass **od**;
perform a pass during which all attribute instances with pass number *m* are evaluated, and as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

Recall that the monotonicity of the attribute evaluation rules and the local safety of the tree transformation rules guarantees that every recomputation of an attribute instance during the transformation phase yields an improvement. Thus, intuitively, Algorithm 4.4 is an attractive alternative to Algorithm 4.1.

We now skip the m-th evaluation pass.

Algorithm 4.5.

repeat
  for i **from** 1 **to** $m-1$
  do perform the i-th evaluation pass **od**;
  perform a pass during which all attribute instances with pass number m are evaluated, and as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

In what follows, we investigate the conditions which guarantee Algorithms 4.4 and 4.5 to have the same input/output behavior and to need the same number of repetitions. Note that in Algorithm 4.4 each tree transformation pass starts with a consistently attributed tree, whereas this is not guaranteed in Algorithm 4.5.

The applicability of tr: (dir, itt, ott, cond, eval) depends on the values of the attribute instances of itt needed as arguments for cond. The separate m-th evaluation pass may be skipped if, during the transformation pass, the necessary attribute instances with pass number m are guaranteed to be evaluated before the applicability of tr is taken into consideration. Observe that no restrictions have to be imposed on the synthesized attribute instances of the new terminals, because condition (b) of Definition 4.4 requires all lexical evaluation rules in eval to compute "correct" values.

Consider the application of a tree transformation rule tr: (**down**, itt, ott, cond, eval) to a subtree with root N. Visiting node N for the first time during the combined m-th evaluation and tree transformation pass the following steps are taken. The first step is the computation of the inherited attribute instances of N (with pass number m). The second step is the possible application of tr. At the beginning of the second step the attribute instances of itt, already computed during this repetition, are:

1) all attribute instances (a, itt, X), for X an arbitrary node of itt, such that pass $(a \textbf{ of } X) \leqq m-1$;

2) all inherited attribute instances (b, itt, Y), for Y the root of itt, such that pass$(b \textbf{ of } Y) = m$.

Hence, with respect to downward tree transformation rule tr, the activities of the m-th evaluation pass may be delayed until the transformation pass, if for every attribute instance (c, itt, Z), not being an inherited attribute instance of the root of itt, and needed as an argument for cond, the following holds: pass$(c \textbf{ of } Z) \leqq m-1$.

We now consider a tree transformation rule tr: (**up**, itt, ott, cond, eval) to be applied to a subtree with root N. During the second visit to node N during the combined m-th evaluation and tree transformation pass the following steps are taken. First, the synthesized attribute instances of N (with pass number

m) are computed. Secondly, the possible applicability of tr is investigated. At the beginning of the second step the values of all attribute instances of itt have a value which was computed during the current repetition.

From these observations we conclude the following theorem.

Theorem 4.2. *Given a simple m-pass attribute grammar with monotonic attribute evaluation rules, and a set of locally safe conditional tree transformation rules. Algorithms 4.4 and 4.5 have the same input/output behavior and need the same number of repetitions, if*:
for every tree transformation rule (**down,** itt, ott, cond, eval), *for every attribute instance* (a, itt, X) *not being an inherited attribute instance of the root of* itt, *and needed as an argument for* cond, *the following holds*: $\text{pass}(a \textbf{ of } X) \leqq m-1$. □

Next, we investigate the combination of the transformation pass and the first subsequent re-evaluation pass, by comparing Algorithms 4.6 and 4.7.

Algorithm 4.6.
perform the first evaluation pass;
repeat
for i **from** 2 **to** m
do perform the i-th evaluation pass **od**;
perform a pass during which all attribute instances with pass number 1 are evaluated, and as many tree transformations are applied as possible;
perform the first evaluation pass
until no tree transformations were applied during the last transformation pass. □

Algorithm 4.7.
perform the first evaluation pass;
repeat
for i **from** 2 **to** m
do perform the i-th evaluation pass **od**;
perform a pass during which all attribute instances with pass number 1 are evaluated and as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

Both algorithms should have the same input/output behavior and need the same number of repetitions.

Observe that the local re-evaluation phase, associated with each tree transformation, includes the recomputation of all the output attribute instances of the complete output template, which may imply that some subtrees, already visited during the combined tree transformation and first re-evaluation pass of Algorithm 4.7 should be visited again to recompute attribute instances with pass number 1.

First, we discuss the consequences of the application of a tree transformation rule tr: (**down,** itt, ott, cond, eval) to a subtree with root N during a downward move.

Assume that ott was extended to compl-ott. Let $X_{q0} \rightarrow X_{q1} \ldots X_{qk} \ldots X_{qn}$ be the production applied immediately above ott and let X_{qk} label the root

of ott. The subtrees already visited during the combined transformation and first re-evaluation pass are the subtrees with root $X_{qj}(1 \leqq j < k, X_{qj} \in V_N)$. The local re-evaluation of compl-ott as part of the application of tr implies a renewed evaluation of the inherited attribute instances $(a, \text{compl-ott}, X_{qj})$ $(1 \leqq j < k$, pass $(a \textbf{ of } X_{qj}) = 1)$, already computed in this pass. This requires another first pass visit to all subtrees with root $X_{qj}(1 \leqq j < k, X_{qj} \in V_N)$ if the local re-evaluator computes for at least one attribute instance $(a, \text{compl-ott}, X_{qj})$ another value than for the corresponding attribute instance $(a, \text{compl-itt}, X_{qj})$. No such problems arise if no extension of ott took place to form compl-ott.

Next, we discuss the consequences of the application of a tree transformation rule tr: (**up,** itt, ott, cond, eval) to a subtree with root N during an upward move.

Again, we consider the case that compl-ott contains an additional production $X_{q0} \to X_{q1} \ldots X_{qk} \ldots X_{qn}$, such that X_{qk} labels the root of ott. The subtrees already visited during the combined pass are the subtrees with root $X_{qj}(1 \leqq j \leqq k, X_{qj} \in V_N)$. Observe that, contrary to the downward case, now also a visit to the restructured subtree has been made. New values for the output attribute instances of compl-ott, being different from the values of the corresponding attribute instances of compl-itt, may require additional first pass visits to both the subtrees with root $X_{qj}(1 \leqq j < k, X_{qj} \in V_N)$ and the subtrees substituted for the variables of ott. Of course, the revision of the first re-evaluation pass may be restricted to the subtrees substituted for the variables of ott if no extension of ott was needed to form compl-ott.

The above-mentioned comparison of corresponding inherited attribute instances of compl-itt and compl-ott has to be done for every possible extension of itt to compl-itt in any derivation tree $T1$ and ott to compl-ott in a derivation tree $T2$ such that $T1[\text{IT}] \xrightarrow{\text{tr}} T2$.

It should be emphasized, however, that only different values for inherited attribute instances may forbid the deletion of a separate first re-evaluation pass. Synthesized attribute instances never cause any problem.

Taking the easy case, with only synthesized attribute instances involved in the first pass, we conclude the following theorem.

Theorem 4.3. *Given a simple m-pass attribute grammar with monotonic attribute evaluation rules, and a set of locally safe conditional tree transformation rules. Algorithms 4.6 and 4.7 have the same input/output behavior and need the same number of repetitions, if:*
all attributes with pass number 1 are synthesized attributes. □

It is easy to see that for attribute grammars and conditional tree transformation rules which obey both criteria formulated in Theorems 4.2 and 4.3 Algorithms 4.8 and 4.9 have the same input/output behavior and need the same number of repetitions.

Algorithm 4.8.

perform the first evaluation pass;
repeat
  for i **from** 2 **to** m
  do perform the i-th evaluation pass **od**;

perform a pass during which all attribute instances with pass numbers 1 and m are evaluated, and as many tree transformations are applied as possible;
perform the first evaluation pass
until no tree transformations were applied during the last transformation pass. □

Algorithm 4.9.
perform the first evaluation pass;
repeat
for i **from** 2 **to** $m-1$
do perform the i-th evaluation pass **od**;
perform a pass during which all attribute instances with pass numbers 1 and m are evaluated, and as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

For a simple 2-pass attribute grammar and a set of conditional tree transformation rules, which obey the above-mentioned criteria, Algorithm 4.9 simplifies to:

Algorithm 4.10.
perform the first evaluation pass;
repeat
perform a pass during which all attribute instances are evaluated and as many tree transformations are applied as possible
until no tree transformations were applied during the last pass. □

An application of the last algorithm will be presented in Sect. 6.2, where tree transformation rules are defined for constant folding, constant propagation and dead code elimination.

5. Comparison with the Evaluation Method for Circular Attribute Grammars

The method of iterating transformation passes over attributed derivation trees, presented in Sect. 4, shows similarities with the evaluation methods for circular attribute grammars, presented by Babich and Jazayeri in [3] and Farrow in [8].

The problem of circular attribute grammars is the impossibility to find for every derivation tree an evaluation order of its attribute instances such that at the moment of execution of every attribute evaluation instruction all the necessary arguments are available.

Babich and Jazayeri [3] solved this problem by supplying *assumptions* for the instances of certain attributes before the evaluation process is started. We will call these attributes the "key" attributes. An assumption is a value for

an instance of a key attribute to be used by the evaluator before it can compute its own value. Every attribute instance, for which no assumption is supplied, should have the property that its value will always be computed before it is needed as an argument in an attribute evaluation instruction.

For circular attribute grammars our solution will be similar to that of Sect. 4. For each attribute a the set $V(a)$ is assumed to be partially ordered with $\perp$ as smallest element. We also assume the attribute evaluation rules to be monotonic.

We initially supply the key attributes with $\perp$ as assumption. Then we may as well assign the value $\perp$ to all attribute instances in the derivation tree, except, of course, to the synthesized attribute instances of the terminals.

Since $T^{\perp}$(i.e., the tree where each attribute instance has its "bottom" value) is locally safe, attribute evaluation will improve the values of the attribute instances in the derivation tree and yield a locally safe tree again (by monotonicity of the evaluation rules). Thus, for any algorithm that repeatedly applies all attribute evaluation instructions, the attributed tree gets better at each step. Hence, if in the domains of the key attributes all ascending chains are finite, the algorithm has to stop, and moreover it will stop with a fixpoint, i.e., with a consistent tree. It is easy to see (by monotonicity again) that it must be the smallest fixpoint.

In general, the evaluation algorithm for a circular attribute grammar may be as follows.

Algorithm 5.1. Evaluation algorithm for a circular attribute grammar with partially ordered attribute domains (such that all ascending chains are finite for the key attributes), and monotonic attribute evaluation rules.

Input: A derivation tree where only the values of the synthesized attribute instances of the leaves are available.

Output: The same derivation tree where all attribute instances have their smallest consistent value.

Algorithm:

assign the value $\perp$ to all instances of the key attributes;
new assumptions:=values of all instances of the key attributes;
repeat
  old assumptions:=new assumptions;
  invoke the evaluator;
  new assumptions:=values of all instances of the key attributes
until new assumptions = old assumptions. □

In this paper it is assumed that the circular attribute grammars under consideration have the simple multi-pass property after the deletion of dependencies $(a, p, j) \rightarrow (b, p, k)$, where a is a key attribute. So, the evaluator to be invoked in Algorithm 5.1 may be a simple multi-pass evaluator.

For compiler optimization purposes a circular attribute grammar may be specified, just to collect all the information necessary for optimization, but without performing any syntactical transformation. This evaluation (information col-

lection) process is repeated until convergence occurs. Subsequently, a single pass over the original derivation tree is made to perform all the possible tree transformations.

The method of iterating transformation and evaluation passes in Sect. 4 resembles this evaluation method for circular attribute grammars for two reasons:

1) The lexical evaluation rules of a tree transformation rule permit the specification of synthesized attribute instances of terminal symbols, which have pass number 0, in terms of attribute instances with pass number >0. In a traditional attribute grammar the inclusion of such a dependency brings along with it a circularity very often.

2) A tree transformation may degrade the consistent value of an attribute instance in a derivation tree to a safe, non-optimal value in a restructured derivation tree, making this attribute instance a candidate for improvement during a subsequent evaluation phase.

In the next section, both the method of iterating mixed transformation and evaluation passes, and the method based on iterative approximations of attribute values (using a circular attribute grammar), followed by a single transformation pass, are applied to the problem of constant folding, constant propagation and dead code elimination for a small programming language.

6. An Example: Constant Folding and Propagation, and Dead Code Elimination

The following example describes constant folding, constant propagation and dead code elimination for a small grammar including assignment, conditional and while statements. The example is borrowed from [16], where global data flow information is collected, used in determining the applicability of optimizing tree transformations, and updated after invalidation of the flow information by tree transformations. The optimization algorithm described in [16] operates on abstract syntax trees, whereas the variants described in this section are defined in terms of concrete derivation trees.

The main topic of this paper is the mixing of attribute evaluation and tree transformation phases, based on the safety of attribute values. However, the evaluation of attribute instances in an unchangeable tree seems more natural. For this reason we first tackle the problem of finding the values of all constant variables and constant expressions everywhere in a derivation tree, by using a traditional yet circular attribute grammar. Having available this information, it can be used to restructure the tree, i.e., to replace all constant variables and constant expressions by constants and to eliminate all dead code.

This approach starts by making passes over the derivation tree to collect data flow information until no more information becomes available. Finally, one transformation pass is performed devoted to the replacement of constant variables and constant expressions by constants and the elimination of dead code. These transformations turn out to keep the derivation tree consistent. This can be checked locally (and also statically) by verifying that compl-itt and compl-ott define the same values for the output attribute instances (cf.

Definition 4.4(c)). So, there is no need to worry about the safety of the tree transformation rules.

Next to this approach, where an attribute evaluation phase is followed by a final and conclusive tree transformation phase, we demonstrate the mixing of attribute evaluation and tree transformation phases.

In Sect. 6.1 the circular attribute grammar approach is discussed. Section 6.2 illustrates the main topic of this paper.

6.1 Specification by a Circular Attribute Grammar

The grammar specifying the collection of data flow information has the following attributes. Associated with each statement is a synthesized attribute mod, which is a finite set of identifier numbers. Attribute mod of a statement includes the identifier numbers of all variables possibly *mod*ified by the statement. Attribute mod is computed in bottom up order, first for assignment statements and then for structured statements.

For constant propagation attributes *i*-pool (*i* for inherited) and *s*-pool (*s* for synthesized) are used. A pool is a finite set of (idno, val) pairs, where idno is the number of an identifier and val its associated value. Inherited attribute *i*-pool of a statement contains the variables which have the same value whenever the execution of the statement is started. Synthesized attribute *s*-pool of a statement includes the variables which have the same value whenever the execution of the statement is finished.

For each assignment statement the following holds. Let idno be the identifier number of the variable in the left part. If the right part is known to be a constant expression with value val, then the pair (idno, val) is inserted into the pool of available constant variables, replacing a pair with the same idno if it exists. If it is unknown whether the right part is a constant expression, then the pair with first component idno (if it exists) is deleted from the pool of available constant variables. In both cases, attribute mod is initialized with a set being composed of idno only.

When leaving a conditional statement an *s*-pool has to be returned which includes those (idno, val) pairs that occur identically in both the *s*-pool of the then part and the else part, unless the value of the condition is known. In this case the *s*-pool of the then part or the else part has to be returned. A similar approach is followed for the computation of mod. The difference is that in case of an unknown condition, the mod values of the then part and the else part are joined.

When entering a while statement, all variables assigned within the while statement have to be deleted from its associated *i*-pool, unless the value of the condition is known to be false, which means that the while statement behaves as a no-operation. The value false for the condition also means that attribute mod must be an empty set. In any other case the value of mod of the while body is passed up.

Associated with every expression and every condition are synthesized attributes intval and boolval, respectively. Both attributes consist of a status field,

indicating whether the expression or condition is known to be constant. If yes, then the second field represents the associated value.

Finally, synthesized attributes idno, intconstval and boolconstval, associated with terminal symbols ident, intconst and boolconst, respectively, are set by the scanner. These attributes are of type number, integer and boolean, respectively.

Attribute grammar AG1 below enumerates the nonterminal and terminal symbols, the start symbol, the attribute descriptions (specifying the attribute types, the association of attributes with grammar symbols and the nature of the attributes, i.e., *inh* for inherited and *syn* for synthesized) and the semantic functions to be used in attribute evaluation rules. The description ends with the productions of the grammar, each followed by its associated set of attribute evaluation rules, enclosed in square brackets. Copy rules between identical attributes of the left-hand side and the right-hand side are deleted in the event of a single nonterminal as the right-hand side of a production.

Attribute Grammar AG1:

nonterminals: program, compound, stats, stat, assignment, condstat, whilestat, cond, expr.

terminals: **begin, end, if, then, else, fi, while, do, od,** :=, +, =, ;, ident, intconst, boolconst.

start symbol: program.

attribute types:

```
const max = ... {maximal number of identifiers allowed in any program to be
                 compiled};
      empty-set-of-ident = [ ];
      empty-pool = [ ];
type  number = 1..max;
      unknown-or-known = (unknown, known);
      inttype =  record
                   case status: unknown-or-known of
                     unknown: ( );
                     known:   (val:integer)
                 end;
      booltype = record
                   case status: unknown-or-known of
                     unknown: ( );
                     known:   (val:boolean)
                 end;
      set-of-ident = set of number;
      pool = set of pool-entry;
      pool-entry = record
                     idno: number,
                     val:  integer
                   end.
```

attributes:

idno: number, **syn of** ident;
intconstval: integer, **syn of** intconst;
boolconstval: boolean, **syn of** boolconst;
intval: inttype, **syn of** expr;
boolval: booltype, **syn of** cond;
mod: set-of-ident, **syn of** compound, stats, stat, assignment, condstat, whilestat;
i-pool: pool, **inh of** compound, stats, stat, assignment, condstat, whilestat, cond, expr;
s-pool: pool, **syn of** compound, stats, stat, assignment, condstat, whilestat.

functions:

function initialize-mod-with (idno: number) **delivers** set-of-ident:
begin {returns the singleton set [idno]} **end**;
function insert (idno: number, intval: inttype) into: (*p*: pool) **delivers** pool:
begin {inserts a new pair (idno, intval.val) into the pool *p*, replacing a pair with the same idno, if existing}
end;
function delete (idno: number) from: (*p*: pool) **delivers** pool:
begin {deletes the pair with first component idno, if existing, from the pool *p*}
end;
function intersect (*p*1, *p*2: pool) **delivers** pool:
begin {returns the pool which is the intersection of the pools *p*1 and *p*2}
end;
function delete-all-identifiers-in (mod: set-of-ident) from: (*p*: pool) **delivers** pool:
begin {deletes all pairs (idno, val) from pool *p* for which idno is in mod}
end;
function element (idno: number) in: (*p*: pool) **delivers** boolean:
begin {checks, whether a pair with first component idno is in pool *p* or not}
end;
function value-of (idno: number) in: (*p*: pool) **delivers** integer:
begin {returns the value associated with idno in pool *p*} **end**.

production rules and semantic rules:

(1) program → compound.
[*i*-pool **of** compound := empty-pool]

(2) compound → **begin** stats **end**.
[mod **of** compound := mod **of** stats;
i-pool **of** stats := *i*-pool **of** compound;
s-pool **of** compound := *s*-pool **of** stats
]

(3) stats[1] → stats[2]; stat.
[mod **of** stats[1] := mod **of** stats[2] + mod **of** stat;

```
        i-pool of stats[2]:=i-pool of stats[1];
        i-pool of stat:=s-pool of stats[2];
        s-pool of stats[1]:=s-pool of stat
      ]
(4)  stats → stat.
(5)  stat → assignment.
(6)  stat → condstat.
(7)  stat → whilestat.
(8)  stat → compound.
(9)  assignment → ident:=expr.
      [mod of assignment:=initialize-mod-with (idno of ident);
       i-pool of expr:=i-pool of assignment;
       s-pool of assignment:=
          if (intval of expr).status=known
          then insert (idno of ident, (intval of expr).val)) into:
                                                    (i-pool of assignment)
          else delete (idno of ident) from: (i-pool of assignment)
          fi
      ]
(10) condstat → if cond then stats[1] else stats[2] fi.
      [mod of condstat:=
          if (boolval of cond).status=unknown
          then mod of statst[1]+mod of stats[2]
          else if (boolval of cond).val=true
               then mod of stats[1]
               else mod of stats[2]
               fi
          fi;
       i-pool of stats[2]:=i-pool of stats[1]:=i-pool of cond:=
                                                    i-pool of condstat;
       s-pool of condstat:=
          if (boolval of cond).status=unknown
          then intersect (s-pool of stats[1], s-pool of stats[2])
          else if (boolval of cond).val=true
               then s-pool of stats[1]
               else s-pool of stats[2]
               fi
          fi
      ]
(11) whilestat → while cond do stats od.
      [mod of whilestat:=
          if (boolval of cond).status=unknown
          then mod of stats
          else if (boolval of cond).val=true
               then mod of stats
               else empty-set-of-ident
               fi
```

```
            fi;
        s-pool of whilestat:=i-pool of stats:=i-pool of cond:=
            if (boolval of cond).status=unknown
            then delete-all-identifiers-in (mod of stats) from:(i-pool of whilestat)
            else if (boolval of cond).val=true
                then delete-all-identifiers-in (mod of stats) from:
                                                        (i-pool of whilestat)
                else i-pool of whilestat
                fi
            fi
        ]
(12) cond→expr[1]=expr[2].
        [i-pool of expr[2]:=i-pool of expr[1]:=i-pool of cond;
         if (intval of expr[1]).status=known and
            (intval of expr[2])·status=known
         then (boolval of cond).status:=known;
              (boolval of cond).val:=
                  ((intval of expr[1]).val=(intval of expr[2]).val)
         else (boolval of cond).status:=unknown
         fi
        ]
        3(13) cond→boolconst.
                [(boolval of cond).status:=known;
                 (boolval of cond).val:=boolconstval of boolconst
                ]
(14) expr[1]→expr[2]+expr[3].
        [i-pool of expr[3]:=i-pool of expr[2]:=i-pool of expr[1];
         if (intval of expr[2]).status=known and
            (intval of expr[3]).status=known
         then (intval of expr[1]).status:=known;
              (intval of expr[1]).val:=
                  (intval of expr[2]).val+(intval of expr[3]).val
         else (intval of expr[1]).status:=unknown
         fi
        ]
(15) expr→ident.
        [if element (idno of ident) in: (i-pool of expr)
         then (intval of expr).status:=known;
              (intval of expr).val:=
                  value-of (idno of ident) in: (i-pool of expr)
         else (intval of expr).status:=unknown
         fi
        ]
(16) expr→intconst.
        [(intval of expr).status:=known;
         (intval of expr).val:=intconstval of intconst
        ]
```

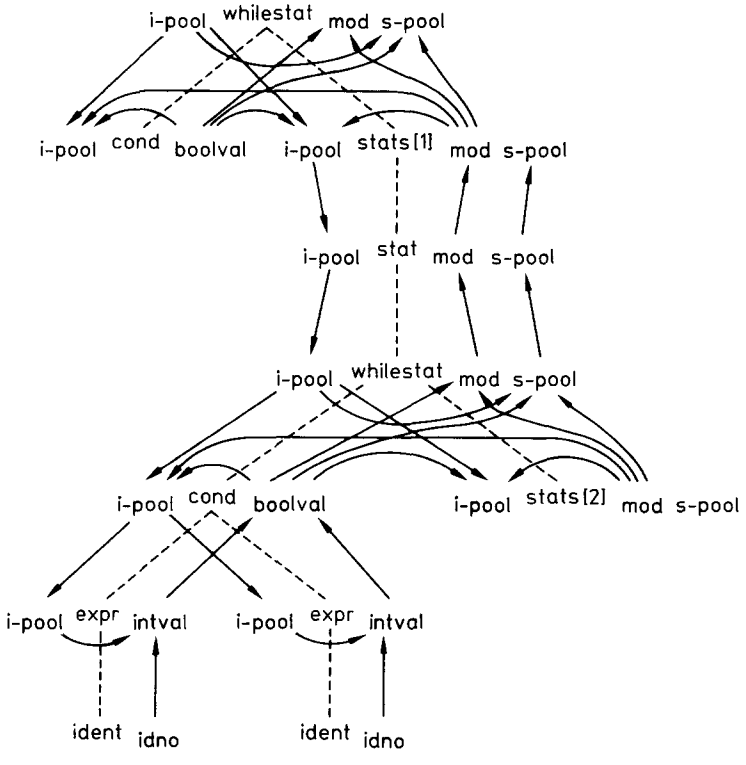

Fig. 4. Circular attribute dependencies

This grammar is ambiguous because of production rule (14). The ambiguity may be resolved by requiring the plus operator to be left associative.

The tree part in Figure 4 shows several circular dependency paths. These circularities can be removed by cutting, for instance, the dependencies between on the one hand attribute occurrence boolval **of** cond and on the other the occurrences of attributes mod and *s*-pool of grammar symbols condstat and whilestat in productions (10) and (11), respectively, and the occurrences of attribute *i*-pool of grammar symbols cond and stats in production (11), i.e., by replacing the used occurrences of (boolval **of** cond).status in the attribute evaluation rules associated with productions (10) and (11) by unknown.

Now the attribute grammar becomes simple 2-pass [1] with the distribution of the attributes over the passes, as shown in Table 6.1.

The synthesized attribute instances of terminal symbols, i.e., intconstval **of** intconst, boolconstval **of** boolconst and idno **of** ident, are assumed to be set by the parser.

The above-mentioned change in the semantic rules of productions (10) and (11) prevents the evaluator from ignoring data flow information from statements which will never be executed. This is a serious loss for constant propagation.

So, we return to the original circular attribute grammar, where we are faced with the problem of needing attribute values which are not yet available. If

Table 6.1. Distribution of the attributes over the passes

Attribute	Pass number
mod	1
intval	2
boolval	2
i-pool	2
s-pool	2

we take the above-mentioned distribution of the attributes over the passes, then the only situation where this will happen is in productions (10) and (11).

For circular attribute grammar AG1 the following theorem is important.

Theorem 6.1. *For the following partial orders:*

— *set inclusion* ($\subseteq$) *on the set of pool values, with the empty set as the smallest element,*

— *its converse* ($\supseteq$) *on the set of mod values, with the set* $\{1, \ldots, \max\}$ *including all identifier numbers as the smallest element,*

— unknown $\leqq$ (known, x) *on the sets of inttype and booltype values, the attribute evaluation rules of attribute grammar* AG1 *are monotonic.*

Proof. From an inspection of the evaluation rules of AG1 it becomes evident that they are monotonic. □

Observe that in the value set of attribute boolval **of** cond all ascending chains are finite (they are of the form: unknown $\leqq$ (known, x)).

To solve the circularity problem of attribute grammar AG1 we supply the assumption "unknown" for all instances of boolval **of** cond in the derivation tree before the evaluation process is started. Every other used attribute occurrence has the property that its value will always be computed before it is needed.

The following theorem gives a fixed upper bound for the number of invocations of the evaluator before the tree is consistently attributed.

Theorem 6.2. *For any program of attribute grammar* AG1, *including W while statements and C conditional statements, each enclosed by a while statement, at most* $W+C+2$ *invocations of the evaluator are needed to find all possible constant expressions.*

Sketch of proof. In general, the number of invocations is at most: (number of key attribute instances) * (length of longest chain $-1)+1$.

For attribute grammar AG1 the chain for attribute boolval **of** cond has length 2. The additional invocation is included to establish convergence. So, the number of invocations is at most: the number of while and conditional statements $+1$.

However, it depends on the embedding of structured statements in while statements whether this number of invocations is really necessary.

The change of status of a conditional or while statement, not enclosed by a while statement, will affect the conditions of other statements during the current execution.

Generally, this is not the case for statements embedded in a while statement. The non-left-to-right dependency between attribute occurrences mod **of** stats and *i*-pool **of** stats of production (11) may cause the effect of the change of the condition of a statement forming part of a while body on the conditions of other statements within the same body to be delayed until the next invocation of the evaluator; a possible effect on the condition of the enclosing while statement will certainly be postponed until the next invocation.

From this we conclude that the number of invocations of the evaluator is at most: 1 {for outer structured statements, i.e., not enclosed by a while statement} $+(W+C)$ {for structured statements embedded in a while statement} $+1$ {for convergence} $= W+C+2$. □

Notice that the non-left-to-right dependencies between on the one hand attribute occurrence mod **of** stats and on the other the occurrences *i*-pool **of** cond and *i*-pool **of** stats in production (11) force the distribution of the attributes over two passes, as shown in Table 6.1, i.e., the execution of the second pass may start as soon as the first pass has been finished. However, there is no non-left-to-right dependency between attributes of the second pass and attributes of the first pass. This means that the $(n+1)$-th execution of the first pass may be performed simultaneously with the n-th execution of the second pass, for any $n \geqq 1$. This observation will be used in the constant folding and propagation part of Algorithm 6.1.

Having found the constant expressions in a derivation tree a single pass over the tree suffices to do all possible transformations.

The following tree transformation rules specify the replacement of a constant expression by a single constant and the elimination of dead code.

transformation rules:

```
trans1: transform down ⟨expr⟩
            cond (intval of expr).status = known
                into ⟨expr, intconst⟩
                    eval intconstval of intconst := (intval of expr).val
        end;
trans2: transform down ⟨condstat, if, cond, then, stats[1], else, stats[2], fi⟩
            cond boolval of cond = (known, true)
                into ⟨compound, begin, stats[1], end⟩
            cond boolval of cond = (known, false)
                into ⟨compound, begin, stats[2], end⟩
        end;
trans3: transform down ⟨whilestat, while, cond, do, stats, od⟩
            cond boolval of cond = (known, true)
                into ⟨loop-forever, forever, do, stats, od⟩
            cond boolval of cond = (known, false)
                into ⟨no-operation⟩
        end.
```

The execution of these transformations during a downward move causes the transformation of a complicated expression or a structured statement to be executed in a single step, whereas bottom up transformations might need several steps.

It is easily verified that for trans1 and trans2 compl-itt and compl-ott define the same values for the output attribute instances. This keeps the tree consistent.

Observe, that any application of rule trans3 will put the derivation tree out of the language defined by the above-mentioned attribute grammar. Of course, a warning should be given if such a tree transformation occurs. To keep the tree in the language one could think of additional productions (and associated attributes and attribute evaluation rules) for a loop-forever and a no-operation. This will be demonstrated in Sect. 6.2.

We are now ready for the complete algorithm that first collects the necessary data flow information and then performs the possible transformations.

Algorithm 6.1. Constant folding and propagation, and dead code elimination according to attribute grammar AG1 and its associated set of tree transformation rules.

Input: A derivation tree where only the synthesized attribute instances of ident, intconst and intbool are available.

Output: A consistently attributed derivation tree where all constant expressions have been replaced by constants and all dead code has been eliminated.

Algorithm:

initialization
assign the value unknown to all instances of attribute boolval **of** cond;
new assumptions:=values of instances of attribute boolval **of** cond;

pre evaluation pass
perform a pass during which the instances of attribute mod are computed;

iteration of evaluation passes
repeat
old assumptions:=new assumptions;
perform a pass during which the instances of all attributes are computed;
new assumptions:=values of instances of attribute boolval **of** cond
until new assumptions = old assumptions;

transformation pass
perform a pass during which all possible transformations are applied. □

The maximal number of passes in this algorithm is expressed in the following corollary.

Corollary 6.1 [of Theorem 6.2] *For any program of attribute grammar* AG1, *including W while statements and C conditional statements, each enclosed by a while statement, at most* $W+C+4$ *passes in Algorithm 6.1 are needed to do all possible constant folding, constant propagation and dead code elimination.* □

6.2 *Specification by Tree Transformation Rules*

In this section we first describe the collection of data flow information by a traditional non-circular attribute grammar AG2, and then enrich this grammar with attributed tree transformation rules to specify constant folding, constant propagation and dead code elimination as well.

Attribute grammar AG2 has attributes idno, intconstval, boolconstval, mod, *i*-pool and *s*-pool, which have the same meaning as the corresponding attributes of AG1. The attribute grammar itself takes a dark view of constant folding and constant propagation in the sense that the values of expressions and conditions are assumed to be unknown, which in fact disallows constant folding and constant propagation. These optimizations are as yet realized by the extension of the attribute grammar with conditional tree transformation rules.

Attribute Grammar AG2:

nonterminals: see AG1, plus: loop-forever and no-operation.
terminals: see AG1, plus: **forever.**
start symbol: program.
attribute types: see AG1, without: unknown-or-known, inttype and booltype.
attributes: see AG1, without: intval and boolval, and
plus: the association of attributes mod, *i*-pool and *s*-pool with loop-forever and no-operation.
functions: see AG1.

production rules and semantic rules:

(1) program → compound.
[*i*-pool **of** compound := empty-pool]
(2) compound → **begin** stats **end.**
[mod **of** compound := mod **of** stats;
i-pool **of** stats := *i*-pool **of** compound;
s-pool **of** compound := *s*-pool **of** stats
]
(3) stats[1] → stats[2]; stat.
[mod **of** stats[1] := mod **of** stats[2] + mod **of** stat;
i-pool **of** stats[2] := *i*-pool **of** stats[1];
i-pool **of** stat := *s*-pool **of** stats[2];
s-pool **of** stats[1] := *s*-pool **of** stat
]
(4) stats → stat.
(5) stat → assignment.
(6) stat → condstat.
(7) stat → whilestat.
(8) stat → compound.
(9) stat → loop-forever.
(10) stat → no-operation.

(11) assignment → ident:=expr.
[mod **of** assignment:=initialize-mod-with (idno **of** ident);
i-pool **of** expr:=*i*-pool **of** assignment;
s-pool **of** assignment:=delete (idno **of** ident) from:
(*i*-pool **of** assignment)
]
(12) assignment → ident:=intconst.
[mod **of** assignment:=initialize-mod-with (indo **of** ident);
s-pool **of** assignment:=insert (idno **of** ident, intval **of** intconst) into:
(*i*-pool **of** assignment)
]
(13) condstat → **if** cond **then** stats[1] **else** stats[2] **fi**.
[mod **of** condstat:=mod **of** stats[1]+mod **of** stats[2];
i-pool **of** stats[2]:=*i*-pool **of** stats[1]:=*i*-pool **of** cond:=
i-pool **of** condstat;
s-pool **of** condstat:=intersect (*s*-pool **of** stats[1],
s-pool **of** stats[2])
]
(14) whilestat → **while** cond **do** stats **od**.
[mod **of** whilestat:=mod **of** stats;
s-pool **of** whilestat:=*i*-pool **of** stats:=*i*-pool **of** cond:=
delete-all-identifiers-in (mod **of** stats) from: (*i*-pool **of** whilestat)
]
(15) loop-forever → **forever do** stats **od**.
[mod **of** loop-forever:=mod **of** stats;
s-pool **of** loop-forever:=*i*-pool **of** stats:=
delete-all-identifiers-in (mod **of** stats) from: (*i*-pool **of** loop-forever)
]
(16) no-operation →.
[mod **of** no-operation:=empty-set-of-ident;
s-pool **of** no-operation:=*i*-pool **of** no-operation
]
(17) cond → expr[1]=expr[2].
[*i*-pool **of** expr[2]:=i-pool **of** expr[1]:=*i*-pool **of** cond]
(18) cond → boolconst.
(19) expr[1] → expr[2]+expr[3].
[*i*-pool **of** expr[3]:=*i*-pool **of** expr[2]:=*i*-pool **of** expr[1]]
(20) expr → ident.
(21) expr → intconst.

This grammar is ambiguous, not only on account of production rule (19), which allows different derivation trees for the same expression, but also because of productions (11), (12) and (21) which allow two derivations for a single integer constant as the right part of an assignment statement.

As for grammar AG1, the first problem can be solved by requiring the plus operator to be left-associative. The second problem will be solved by giving priority to the combination of productions (11) and (21) in the case of the assignation of a constant value. In fact, the parser does not know production

rule (12). This rule has been included for optimization purposes only and the integer constant in the right part of production (12) is assumed to be compiler-made. As a matter of fact, the integer constant has to be hoisted into a production for the assignment statement to make its attribute intconstval visible, because one of the basic concepts of attribute grammars is that the evaluation rules are associated with productions only. Instances of attributes farther away in the tree are invisible.

Attribute grammar AG2 is simple 2-pass [1] with the same distribution of the attributes mod, *i*-pool and *s*-pool over the passes, as shown in Table 6.1. Again the instances of idno, inconstval and boolconstval are assumed to be set by the parser.

The following tree transformation rules specify the conditional replacement of a variable by a constant, constant folding, and dead code elimination.

transformation rules:

```
trans1: transform up ⟨expr, ident⟩
            cond element (idno of ident) in: (i-pool of expr)
                into ⟨expr, intconst⟩
                    eval intconstval of intconst:=value-of (idno of ident) in:
                                                        (i-pool of expr);
        end;
trans2: transform up ⟨expr, ⟨expr, intconst[1]⟩, +, ⟨expr, intconst[2]⟩⟩
            into ⟨expr, intconst⟩
                eval intconstval of intconst:=intconstval of intconst[1] +
                                              intconstval of intconst[2]
        end;
trans3: transform up ⟨cond, ⟨expr, intconst[1]⟩, =, ⟨expr, intconst[2]⟩⟩
            into ⟨cond, boolconst⟩
                eval boolconstval of boolconst:=(intconstval of intconst[1]
                                                 =
                                                 intconstval of intconst[2])
        end;
trans4: transform up ⟨assignment, ident, :=, ⟨expr, intconst⟩⟩
            into ⟨assignment, ident, :=, intconst⟩
        end;
trans5: transform up ⟨condstat, if, ⟨cond, boolconst⟩,
                                then, stats[1], else, stats[2], fi⟩
            cond boolconstval of boolconst = true
                into ⟨compound, begin, stats[1], end⟩
            cond boolconstval of boolconst = false
                into ⟨compound, begin, stats[2], end⟩
        end;
```

```
trans6: transform up ⟨whilestat, while, ⟨cond, boolconst⟩, do, stats, od⟩
            cond boolconstval of boolconst = true
                into ⟨loop-forever, forever, do, stats, od⟩
            cond boolconstval of boolconst = false
                into ⟨no-operation⟩
        end.
```

For safety considerations the following theorems are important.

Theorem 6.3. *For the following partial orders:*

— *set inclusion* ($\subseteq$) *on the set of pool values, with the empty set as the smallest element,*

— *its converse* ($\supseteq$) *on the set of mod values, with the set* $\{1, \ldots, \max\}$ *including all identifier numbers as the smallest element,*

the attribute evaluation rules of attribute grammar AG2 *are monotonic.*

Proof. Easily verified by checking the evaluation rules of AG2. □

Theorem 6.4. *The tree transformation rules* trans1 *through* trans6 *are locally safe.*

Proof. The tree transformation rules meet all the conditions of Definition 4.4.

Take, for example, transformation rule trans5. We check condition (*c*). To form compl-itt and compl-ott, itt and ott have to be extended with the productions stat → condstat and stat → compound, respectively. We discuss the case that boolconstval **of** boolconst has the value true. Observe that mod **of** stat will decrease, *i*-pool **of** stats[1] will stay the same and *s*-pool **of** stat will increase as a result of the transformation. Hence, the values of all the output attribute instances of compl-ott improve. □

Every tree transformation may open up the applicability of further transformations. To let each tree transformation rule benefit from earlier transformations as soon as possible, the rules are applied in bottom up order. Transformation rule trans1 is the only one where the direction makes no difference (because the transformation happens at the bottom of the tree).

Notice that for the extended attribute grammar AG2 the n-th execution of the transformation pass may be combined with the n-th execution of the second evaluation pass (since the direction of all tree transformations is **up**) and the $(n+1)$-th execution of the first evaluation pass (since this pass works strictly bottom up), for any $n \geqq 1$ (Theorems 4.2 and 4.3, and Algorithm 4.10).

This observation leads to the following algorithm for constant folding and propagation, and dead code elimination according to attribute grammar AG2.

Algorithm 6.2. Constant folding and propagation, and dead code elimination according to attribute grammar AG2 and its associated set of tree transformation rules.

Input: A derivation tree T where only the synthesized attribute instances of ident, intconst and intbool are available.

Output: An attributed derivation tree, consistently derivable from T^c, where all constant expressions have been replaced by constants and all dead code has been eliminated.

Algorithm:

pre evaluation pass

perform a pass during which the instances of attribute mod are computed;

iteration of evaluation and transformation passes

repeat

perform a pass during which all attribute instances are evaluated and as many tree transformations are applied as possible

until no tree transformation rules were applied during the last pass. □

The maximal number of passes in this algorithm is expressed in the following theorem.

Theorem 6.5. *For any program of attribute grammar* AG2, *including W while statements and C conditional statements, each enclosed by a while statement, at most* $W+C+3$ *passes in Algorithm 6.2 are needed to do all possible constant folding, constant propagation and dead code elimination.*

Proof. See Theorem 6.2 and Corollary 6.1. □

6.3 An Example of the Example

The following program shows an example where the repetition of the combined tree transformation and attribute evaluation pass leads to further improvements.

begin
$a:=2; b:=1; c:=1;$
while $a=b$ **do if** $b=c$ **then** $d:=1$ **else** $a:=1$ **fi od**
end.

In Algorithm 6.2, the first execution of the combined tree transformation and attribute evaluation pass results in the replacement of the conditional statement by its then part. During the second execution the while statement is replaced by a no-operation. No more tree transformations are performed during the third execution.

The resulting program is

begin $a:=2; b:=1; c:=1;$ **end.**

The same number of iterations is needed if the circular attribute grammar is applied. In Algorithm 6.1, the first iteration produces the value **true** for the condition of the conditional statement. The second iteration results in the value **false** for the while condition. The third iteration establishes convergence.

Having available all the necessary data flow information, a single pass over the derivation tree is now sufficient to do all the possible tree transformations, giving rise to the same program as found by the method where tree transformations and re-evaluations are performed in parallel.

7. Discussion

An implementation of compiler optimizations is discussed where conditional tree transformations are performed during a pass over a derivation tree, which is never interrupted for re-evaluation purposes. This is certainly allowed for transformations which guarantee the attribute instances in the derivation tree to remain unaffected. If not, then a distinction is made between consistent and safe attribute values, both correct and excluding incorrectly applied tree transformations. This allows the transformation algorithm to proceed, possibly at the price of missing some transformations during the current pass. Safe attribute values also allow the combination of attribute evaluation and tree transformation phases.

An alternative is the formulation of a circular attribute grammar which specifies a complete evaluation of both the original derivation tree and the tree as it should be after its reconstruction. After the completion of all precomputations a single final pass suffices to do all transformations.

The advantage of the circular attribute grammar approach is that less unnecessary pattern matching and computation of enabling conditions has to be done. The disadvantage is that generally more space and time are needed for additional attributes and associated computations. Moreover, an additional pass is needed.

A different approach is the application of an optimal global re-evaluation phase after every tree transformation, which minimizes the number of recomputations and the number of tree traversals (cf. [2]). The advantage of taking full profit of consistent attribute values is that generally less transformation passes are needed. The disadvantage is the need of additional attributes for bookkeeping purposes.

Machine-independent optimizations form an essential part of a compiler writing system being developed at the University of Twente. Each of the three above-mentioned strategies is being implemented. Comparative experiments have to show whether one of these should be given preference above the others.

Acknowledgements. Thanks go to Joost Engelfriet for his valuable and critical comments, which considerably improved this paper.

References

1. Alblas, H.: A characterization of attribute evaluation in passes. Acta Inf. **16**, 427–464 (1981)
2. Alblas, H.: Optimal incremental simple multi-pass attribute evaluation. Memorandum INF-86-27, Department of Informatics, University of Twente (1986)
3. Babich, W.A., Jazayeri, M.: The method of attributes for data flow analysis. Part I. Exhaustive analysis. Acta Inf. **10**, 245–264 (1978)

4. Bochmann, G.V.: Semantic evaluation from left to right. Commun. ACM **19**, 55–62 (1976)
5. Chirica, L.M., Martin, D.F.: An order-algebraic definition of Knuthian semantics. Math. Syst. Theory **13**, 1–27 (1979)
6. DeRemer, F.L.: Transformational grammars. In: Bauer, F.L., Eickel, J. (eds.) Compiler construction: An Advanced Course. (Lect. Notes Comput. Sci., Vol. 21, pp. 121–145) Berlin Heidelberg New York: Springer 1974
7. Engelfriet, J.: Attribute grammars: Attribute Evaluation Methods. In: Lorho, B. (ed.) Methods and Tools for Compiler Construction, pp. 103–138. Cambridge: Cambridge University Press 1984
8. Farrow, R.: Automatic generation of fixed-point finding evaluators for circular, but well-defined, attribute grammars. In: Proc. SIGPLAN 1986 Symposium on Compiler Construction. SIGPLAN Notices **21**, 85–98 (1986)
9. Ganzinger, H., Giegerich, R.: A truly generative semantics directed compiler generator. In: Proc. SIGPLAN 1982 Symposium on Compiler Construction. SIGPLAN Notices **17**, 172–184 (1982)
10. Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximative semantics with respect to program transformations. In: Informatik-Fachberichte 50, pp. 1–10. Berlin Heidelberg New York: Springer 1981
11. Glasner, I., Möncke, U., Wilhelm, R.: OPTRAN, a language for the specification of program transformations. In: Informatik-Fachberichte 34, pp. 125–142. Berlin Heidelberg New York: Springer 1980
12. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**, 127–145 (1968). Correction in: Math. Syst. Theory **5**, 95–96 (1971)
13. Möncke, U., Weisgerber, B., Wilhelm, R.: How to implement a system for manipulation of attributed trees. In: Informatik Fachberichte 77, pp. 112–127. Berlin Heidelberg New York: Springer 1984
14. Nestor, J.R., Mishra, B., Scherlis, W.L., Wulf, W.A.: Extensions to attribute grammars. Techn. Rep. TL 83-36, Tartan Laboratories 1983
15. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language based editors. ACM Trans. Progr. Lang. **5**, 449–477 (1983)
16. Wilhelm, R.: Computation and use of data flow information in optimizing compilers. Acta Inf. **12**, 209–225 (1979)
17. Yeh, D.: On incremental evaluation of ordered attribute grammars. BIT **23**, 308–320 (1983)

Received December 3, 1987 / April 10, 1989