OPTIMAL INCREMENTAL SIMPLE MULTI-PASS ATTRIBUTE EVALUATION

Henk ALBLAS

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

Communicated by R. Wilhelm
Received 27 April 1989
Revised 19 June 1989

Keywords: Attribute grammars, incremental multi-pass evaluation, tree transformations

1. Introduction

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

A tree transformation may invalidate attribute instances, not only in the restructured part of the tree but also elsewhere in the tree. To make the attribution of a derivation tree correct again (which is generally needed in order to be able to test the predicates of subsequent tree transformations), a re-evaluation of the entire tree could be applied. However, a repeated computation of all the attribute instances after each transformation is inefficient and should be avoided (cf. [7,9,10,12]). In this paper an incremental simple multi-pass evaluator is presented, which works optimally in the number of visits to tree nodes and the number of recomputations. By optimal we mean that, whenever possible, subtrees are skipped, and no unnecessary calculations are done. Optimization of the non-incremental simple multi-pass attribute evaluator by skipping subtrees is discussed in [8].

2. Notation and conventions

An attribute grammar [6] is a context-free grammar, augmented with attributes and attribute evaluation rules. Each grammar symbol $X$ has a finite set $A(X)$ of attributes, partitioned into disjoint subsets $I(X)$ and $S(X)$ of inherited and synthesized attributes, respectively. The start symbol has no inherited attributes. The set of all attributes will be denoted by $A$. An attribute $a$ of symbol $X$ is also denoted by $a$ of $X$.

A production $p : X_{p_0} \rightarrow X_{p_1} X_{p_2} \ldots X_{p_n}$ is said to have attribute occurrence $(a, p, k)$ if $a \in A(X_{p_k})$. The set of attribute occurrences of production $p$ can be partitioned into two disjoint sets of defined occurrences and used occurrences. The defined occurrences are the synthesized occurrences of the left-hand side and the inherited occurrences of the right-hand side. The used occurrences are the inherited occurrences of the left-hand side and the synthesized occurrences of the right-hand side. Associated with each production $p$ is a set of attribute evaluation rules which specify how to compute the values of its
defined attribute occurrences in terms of the values of its used attribute occurrences. We say that \((b, p, k)\) depends on \((a, p, j)\) if \((a, p, j)\) is an argument in the evaluation rule for \((b, p, k)\).

Given a derivation tree, instances of attributes are attached to the nodes in the following way: if node \(N\) is labelled 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 \(N\).

3. Simple multi-pass evaluation

Throughout this paper we assume that the attribute evaluation strategy is simple multi-pass (see e.g. [1]), which means that a fixed number of depth-first left-to-right and/or right-to-left traversals (called passes) are made over the derivation tree and all instances of the same attribute are evaluated during the same pass. The directions of the successive passes are indicated by a sequence \(\langle d_1, \ldots, d_m \rangle\) where \(d_i\) (1 \(\leq i \leq m\)) denotes the direction of the \(i\)th pass, which is either L (left-to-right) or R (right-to-left).

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 with respect to a given sequence of pass directions \(\langle d_1, \ldots, d_m \rangle\) if \(A_0\) consists of the synthesized attributes of the terminal symbols and the instances of all attributes in \(A_i\) (1 \(\leq i \leq m\) can be evaluated during the \(i\)th pass of the 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 with respect to some sequence of pass directions \(\langle d_1, \ldots, d_m \rangle\). 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 \(pass: A \rightarrow \{0, 1, \ldots, m\}\) can be defined as \(pass(a) = i\) if \(a \in A_i\).

The following tree-walk algorithm (cf. P3 of [3]), defines a simple multi-pass evaluator. Node \(N\) is denoted by either “\(N.0\)” or “\(N\)” and “\(k\)th son of \(N\)” is written as “\(N.k\)” The father of \(N\) is written as “father\((N)\)” and “brother-number\((N)\)” denotes the integer \(k\) such that \(N = \text{father}(N).k\).

**Algorithm 1. Simple multi-pass evaluation.**

**Input:** An attributed derivation tree \(T\) in which only the synthesized attribute instances of the terminal symbols are defined, a sequence of pass directions \(\langle d_1, \ldots, d_m \rangle\), and a partition \(\langle A_0, A_1, \ldots, A_m \rangle\) of the set of attributes \(A\) which is correct with respect to \(\langle d_1, \ldots, d_m \rangle\).

**Output:** The attributed derivation tree \(T\) in which all attribute instances are defined.

**Algorithm:**

```plaintext
begin
    const m = ...;
    type node = ...;
    pass number = 1..m;
    pass direction = (L,R);
    var N, root: node;
    i, n: integer;
    pass finished: boolean;

    function successor (k, n: integer; d: pass direction): integer;
        begin successor :=
            if d = L then (k+1) mod (n+1) else (k-1) mod (n+1)
        end [of successor];

    read(root);
    for i from 1 to m
```
do $N := \text{root}; k := 0;$

$\text{pass finished} := \text{false};$

while not $\text{pass finished}$

do (let $p: \mathcal{X}_{p0} \rightarrow \mathcal{X}_{p1}\mathcal{X}_{p2}\ldots\mathcal{X}_{pn}$ be the production applied at $N$)

$k := \text{successor}(k, \mathcal{N}, d_i);$ 

if $k = 0$

then for all $a \in I(X_{pk})$

\hspace{1em} do if $a \in A_i$

\hspace{2em} then evaluate $a$ of $N, k$

\hspace{2em} if \od;

\hspace{1em} if $X_{pk}$ is a nonterminal symbol

\hspace{2em} then $N := N, k; \ k := 0$

else for all $a \in \mathcal{S}(X_{pk})$

\hspace{2em} do if $a \in A_i$

\hspace{3em} then evaluate $a$ of $N$

\hspace{3em} if \od;

\hspace{1em} if $N \neq \text{root}$

\hspace{2em} then $k := \text{brother-number}(N); \ N := \text{father}(N)$

else $\text{pass finished} := \text{true}$

\hspace{2em} if \fi

\od

end.

4. Conditional tree transformations

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

A tree template is a possibly incomplete derivation tree (incomplete in the sense that arbitrary symbols may label the root and the leaves). Nonterminal symbols labelling 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 consists of an input tree template $\text{itt}$ and an output tree template $\text{ott}$. Context conditions can be expressed by enabling conditions which are predicates on attribute instances of $\text{itt}$.

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

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

We denote a conditional tree transformation rule by $\text{tr}: (\text{itt, ott, cond, eval})$, where $\text{itt}$ and $\text{ott}$ are the input and the output tree template, $\text{cond}$ is the enabling condition and $\text{eval}$ is the set of lexical evaluation rules.
A conditional tree transformation rule \( tr: (itt, ott, cond, eval) \) is applicable to a subtree \( IT \) of an attributed derivation tree \( T_1 \), if \( IT \) is an instance of \( itt \) and the evaluation of \( cond \) yields true. The application of \( tr \) consists of the creation of an instance \( OT \) of \( ott \) (in which the correspondence between subtrees and variables, established by \( IT \) and \( itt \), is maintained) and the replacement of \( IT \) by \( OT \), thus creating a (partially attributed) derivation tree \( T_2 \). Moreover, it is assumed that the inner and the output attribute instances of \( ott \) are given the value \emph{unknown}, and that the values of the synthesized attribute instances associated with the new terminal nodes of \( ott \) are computed using the rules specified by \( eval \). Such an attributed derivation tree \( T_2 \) may contain incorrect attribute values everywhere in \( T_2 \), because of long-reaching attribute dependencies. A tree transformation may even cause the values of the input attribute instances of \( ott \) to be incorrect.

5. A re-evaluation strategy

We present a re-evaluator which works optimally in the number of visits to tree nodes and the number of re-evaluations of attribute instances. By optimal we mean that, whenever possible, subtrees are skipped, and that the only attribute instances subject to re-evaluation are the attribute instances having an incorrect value and the attribute instances directly depending on these attribute instances. Since the simple multi-pass strategy is used, it follows that the recomputation of each attribute instance is considered once, and only after its argument values have been reconsidered. Hence, it is guaranteed that the value of every attribute instance is correct after the completion of the re-evaluation process. Moreover, we restrict the re-evaluation passes to the smallest possible subtree surrounding the restructured subtree. This ensures, in particular, that the re-evaluator works in time and space linear in the size of the “affected area” of the tree, i.e., all those nodes that have at least one wrong attribute value or at least one attribute value depending on a wrong value.

To be able to mark the attribute instances that need to be evaluated, we associate with every tree node a variable \( \text{NeedToBeEvaluated} \) of type set of attributes. For \( \text{NeedToBeEvaluated} \) associated with node \( N \) we use the same notation as for attributes, namely \( \text{NeedToBeEvaluated} \) of \( N \). To update \( \text{NeedToBeEvaluated} \) properly, we introduce a variable \( \text{Changed} \) of type set of attributes. During a pass, a node will be visited downwards and upwards. In a downward visit of a node the variable \( \text{Changed} \) includes the inherited attribute instances of the node which have changed their value during the current pass. Similarly, in an upward visit \( \text{Changed} \) includes the synthesized attribute instances of the node whose values have changed during the current pass. Attribute \( a \) is inserted in \( \text{NeedToBeEvaluated} \) of \( N \) as soon as an argument of the attribute evaluation rule of \( a \) of \( N \) has changed, as indicated by variable \( \text{Changed} \). Deletion is done immediately after the recomputation of \( a \) of \( N \).

To improve the tree-walk strategy we associate with every tree node labelled by a nonterminal symbol a variable \( \text{SubtreeAffected} \) of type set of pass numbers. Let \( N_0 \) be a node, \( p: X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn} \) the production applied at \( N_0 \), and \( N_1, N_2, \ldots, N_n \) the sons of \( N_0 \) from left to right, respectively. \( \text{SubtreeAffected} \) of \( N_j \) contains pass number \( i \) if and only if either a defined attribute occurrence \((a, p, j)\) exists, such that pass\((a \text{ of } X_{pj}) = i \) and \( a \in \text{NeedToBeEvaluated} \) of \( N_j \), for some \( j (0 < j < n) \), or \( i \in \text{SubtreeAffected} \) of \( N_j \) for some \( j (1 < j < n) \).

During re-evaluation passes, \( \text{SubtreeAffected} \) of \( N_0 \) is updated when \( N_0 \) is visited for the second time during a pass.

This scheme guarantees a correct value for \( \text{SubtreeAffected} \) of \( N_0 \) whenever the re-evaluator is not in the subtree with root \( N_0 \). This makes it possible to skip the subtree with root \( N_0 \) whenever the re-evaluator returns to \( N_0 \) during a pass with number \( i \), where \( i \not\in \text{SubtreeAffected} \) of \( N_0 \). At the end of the re-evaluation process \( \text{SubtreeAffected} \) of \( N_0 \) is empty for all \( N_0 \).

The re-evaluator starts its first pass at the root of the restructured subtree, which we call the “subtree.
under consideration”. If, at the end of a pass over the subtree under consideration, the value of Changed turns out to be non-empty, then the current subtree under consideration is widened, i.e., the father of its root becomes the root of the new subtree under consideration. The pass is then continued over the widened tree. This process of widening continues until the value of Changed is empty. Every subsequent pass starts at the root of the current subtree under consideration.

Immediately before the application of a tree transformation rule tr: (itt, oft, cond, eval) to a derivation tree T1, the values of all attribute instances of T1 are correct. Also, for every node N of T1, NeedToBeEvaluated of N and SubtreeAffected of N are empty. Since the values of the inner and the output attributes are unknown, it follows that, as a consequence of a tree transformation, the following actions must be taken before the re-evaluator may be activated.

1. Every inner and every output attribute instance of oft must be included in its corresponding set NeedToBeEvaluated.
2. For every non-leaf node N of oft its associated variable SubtreeAffected of N must be set. From the fact that SubtreeAffected of N is defined in terms of variables associated with N itself and its immediate descendants, it follows that the instances of SubtreeAffected associated with the non-leaf nodes of oft can easily be evaluated from the bottom up.

Simple multi-pass re-evaluation using this scheme is defined in Algorithm 2. In this algorithm we use the statement “re-evaluate a of N”, by which we mean the following steps:

\[
\begin{align*}
\text{old} & := a \text{ of } N; \\
\text{evaluate } a \text{ of } N; \\
\text{new} & := a \text{ of } N; \\
\text{if } \text{old} \neq \text{new} & \text{ then insert } a \text{ in Changed }; \\
\text{delete } a & \text{ from NeedToBeEvaluated of } N.
\end{align*}
\]

Algorithm 2 has been adapted from Algorithm 1. Procedure “propagate change” (with parameters k of type integer and p of type production number) inserts attribute instances, for which one of the arguments is found in Changed, in their corresponding sets NeedToBeEvaluated. For k = 0, Changed contains inherited attribute instances associated with the left-hand side of a production, and for k ≠ 0, Changed contains synthesized attribute instances associated with the kth symbol of the right-hand side.

**Algorithm 2.** Simple multi-pass re-evaluation.

**Input:** An attributed derivation tree T in which all attribute instances have a value but some attribute instances may have an incorrect value, the (properly initialized) sets NeedToBeEvaluated and SubtreeAffected associated with tree nodes, a sequence of pass directions \(\langle d_1, \ldots, d_m \rangle\), and a partition \(\langle A_0, A_1, \ldots, A_m \rangle\) of the set of attributes A which is correct with respect to \(\langle d_1, \ldots, d_m \rangle\).

**Output:** The attributed derivation tree T in which all attribute instances have their correct values.

**Algorithm:**

```plaintext
begin
const EmptySetOfAttributes = [];
m = ...;
type node = ...;
production number = 1..r;
pass number = 1..m;
pass direction = (L,R);
attributes = (... enumeration of attributes ...);
set of attributes = set of attributes;
var N, root of main tree,
root of subtree under consideration: node;
i: pass number;
k, n: integer;
p: production number;
```

293
pass finished: boolean;
Changed: set of attributes;

function successor (k, n: integer; d: pass direction): integer:
begin (as in Algorithm 1) end;

procedure propagate change (k: integer; p: production number);
begin (let $p: X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$ be the production with number $p$)
if Changed $\neq$ EmptySetOfAttributes
then for $j$ from 0 to $n$
do for all $a \in A(X_{p0})$
do for all $b \in A(X_{pk})$
do if $(a,p,j)$ depends on $(b,p,k)$ and $b \in Changed$
then insert $a$ in NeedToBeEvaluated of $N_j$
fi
od
od
od
fi
end (of propagate change);

read(root of main tree); read(root of subtree under consideration);
for $i$ from 1 to $m$
do if $i \in SubtreeAffected of root of subtree under consideration$
then Changed := EmptySetOfAttributes;
N := root of subtree under consideration; $k := 0$;
pass finished := false;
while not pass finished
do (let $p: X_{p0} \rightarrow X_{p1}X_{p2}...X_{pn}$ be the production at $N$
propagate change($k$, $p$);
Changed := EmptySetOfAttributes;
k := successor($k$, $n$, $d_i$);
if $k \neq 0$
then [downward visit of $N.k$]
for all $a \in I(X_{pk})$
do if $a \in A_j$ and $a \in NeedToBeEvaluated of N.k$
then re-evaluate $a$ of $N.k$
fi
od:
if $X_{pk}$ is a nonterminal symbol
then if Changed $\neq$ EmptySetOfAttributes
or $i \in SubtreeAffected of N.k$
then $N := N.k$; $k := 0$
fi
fi
else (upward visit of $N$)
for all $a \in S(X_{pk})$
do if $a \in A_j$ and $a \in NeedToBeEvaluated of N$
then re-evaluate $a$ of $N$
fi
od;
SubtreeAffected of $N := \{ pass(a of X_{pj}) \mid 0 \leq j \leq n.$
$(a,p,j)$ is a defined attribute occurrence,
and $a \in NeedToBeEvaluated of N.j\}$;
for $j$ from 1 to $n$
do if $X_{pj}$ is a nonterminal symbol
then SubtreeAffected of $N :=$ SubtreeAffected of $N \cup SubtreeAffected of N.j$
fi
od:
if $N =$ root of subtree under consideration
and $N =$ root of main tree
and Changed $\neq$ EmptySetOfAttributes
then root of subtree under consideration := father($N$)
fi:
if $N =$ root of subtree under consideration
Algorithm 2 works in time linear in the size of the affected area of the tree since each attribute instance in the affected area is recomputed at most once, and at most \( m \) passes are made over any part of the affected area. The space required for bookkeeping information (one bit for every attribute or pass number in every instance of \text{NeedToBeEvaluated} and \text{SubtreeAffected}) is also linear in the size of the affected area.

The approach used in Algorithm 2 can easily be adjusted to the visit sequences of ordered attribute grammars \([4,5]\), for which we also need variables \text{NeedToBeEvaluated} and \text{SubtreeAffected}. The difference with incremental simple multi-pass evaluation is that for ordered attribute grammars the variables \text{SubtreeAffected} are of type set of visit numbers (for more details see [2]). A similar method is used by Engelfriet in [3], although he does not make use of sets \text{NeedToBeEvaluated} and \text{Changed}. A tree node is marked as affected for all future visits if one of its attribute instances changes its value. This may lead to unnecessary re-evaluations and unnecessary visits to subtrees, although time and space of this re-evaluator are also linear in the size of the affected area. Another (quite involved) solution for ordered attribute grammars is presented in [12]. An incremental evaluation algorithm for absolutely non-circular attribute grammars can be found in [9]. Solutions for arbitrary non-circular attribute grammars are discussed in [7,9,10]. It is not surprising that the algorithms for the wider class of arbitrary non-circular attribute grammars turn out to be more complicated than Algorithm 2.

Although we discussed incremental attribute evaluation in the context of optimizing tree transformations. Algorithm 2 works for any derivation tree with wrong attribute values. Hence, our incremental simple multi-pass attribute evaluator works for incremental editing as well.

Acknowledgment

I wish to thank Joost Engelfriet for his valuable and critical comments.

References