Acta Informatica 24, 299–352 (1987)

Acta Informatica

© Springer-Verlag 1987

One-Pass Transformations of Attributed Program Trees

Henk Alblas
University of Twente, Department of Informatics, P.O. Box 217
NL-7500 AE Enschede, The Netherlands

Contents

Summary . . . 299
1. Introduction . . . 300
2. Basic Concepts . . . 301
3. Conditional Tree Transformations . . . 303
4. Dependencies Between Attributes Involved in Successive Transformations . . . 309
5. An Algorithm to Test the Safety of Successive Transformations . . . 315
5.1 A First Investigation: Necessary Graphs . . . 316
5.2 Dependencies Between Attributes of a Single Tree Node . . . 324
5.3 Dependencies Between Attributes of Two Tree Nodes . . . 330
5.4 Context Dependencies for Tree Templates . . . 337
5.5 The Safety Test . . . 338
5.6 Comparison of Methods . . . 341
6. An Example: Constant Propagation and Dead Code Elimination . . . 343
6.1 Collection of Data Flow Information . . . 343
6.2 Transformation Rules . . . 347
6.3 Iterative Application of a Transformation Pass . . . 350
References . . . 352

Summary. The classical attribute grammar framework can be extended by allowing the specification of tree transformation rules. A tree transformation rule consists of an input template, an output template, enabling conditions which are predicates on attribute instances of the input template, and re-evaluation rules which define the values of attribute instances of the output template. A tree transformation may invalidate attribute instances which are needed for additional transformations.

In this paper we investigate whether consecutive tree transformations and attribute re-evaluations are safely possible during a single pass over the derivation tree. This check is made at compiler generation time rather than at compilation time.

---

* Part of this work was done while the author was visiting Tartan Laboratories Inc., Pittsburgh, PA, USA

A graph theoretic characterization of attribute dependencies is given, showing in which cases the recomputation of attribute instances can be done in parallel with tree transformations.

1. Introduction

Attribute grammars have proved to be useful in describing programming languages, their editors and compilers. In this paper we will consider the application of attribute grammars for the specification of optimizing tree transformations.

In the classical attribute grammar framework [17], a set of attribute evaluation rules is associated with every production. These rules specify how to compute the values of certain attribute occurrences as a function of other attribute occurrences.

For the specification of tree transformations an extension [11, 21, 25] of the attribute grammar framework with tree transformation rules is needed. Such a tree transformation rule includes: a description of the input template (i.e., the structure of the part of the derivation tree to which the transformation has to be applied), a description of the output template (i.e., the structure of the transformed part of the derivation tree), enabling conditions which are predicates on attribute instances of the input template, and re-evaluation rules which explicitly define the values of those attribute instances of the output template that cannot be taken over from the input template.

When starting the tree transformation process all attribute instances attached to the derivation tree are assumed to be available, i.e., evaluated. 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 evaluation rules will deliver different values.

To make the attribution of a derivation tree correct again (which is needed in order to be able to test the enabling conditions of the next tree transformation), a re-evaluation of the whole tree could be applied. However, a repeated computation of all attribute instances after each transformation is time consuming and should be avoided.

To minimize work, the re-evaluation process could be confined – as much as possible – to those attribute instances whose values became incorrect by transforming the derivation tree. This approach is followed by Möncke c.s. in [19] and Alblas in [3] for optimizing tree transformations, and also by Reps c.s. in [6, 23, 24], Yeh in [26] and Engelfriet in [8] for syntax directed editing, where the decisions to re-evaluate attribute instances are made on the fly, i.e., at compile time.

A different approach is followed in this paper. We propose that all tree transformations be performed during a single pass over the derivation tree. This means that the re-evaluation process after each tree transformation can be confined to the incorrect attribute instances which are met during the continuation of the transformation pass and are needed for the enabling conditions of later transformations and for the arguments of their re-evaluation rules.

A prerequisite for the application of this strategy is to know in advance (i.e., at compiler construction time rather than at compile time) whether for

any derivation tree and for any sequence of tree transformations it is possible to perform the transformations and re-evaluations during a single pass.

With respect to the attribute instances involved in enabling conditions and re-evaluation rules, we investigate their dependencies statically to find out whether the recomputation of possibly incorrect but necessary attribute instances can be done correctly and in time during the transformation pass, i.e., without interrupting the pass to make extra tree traversals for re-evaluation purposes.

A graph theoretic characterization of attribute dependencies is given, showing in which cases the recomputation of the required attribute instances can be done in parallel with the tree transformations.

This paper is organized as follows. Section 2 provides an introduction to the basic concepts associated with attribute grammars. In Sect. 3 we discuss the concept of a conditional tree transformation rule, and the idea of performing tree transformations and attribute re-evaluations during a single pass over a derivation tree. Section 4 describes the attribute dependencies which may prevent the correct recomputation of attribute values between successive tree transformations. In Sect. 5 algorithms are developed which test whether the dependencies between attribute instances involved in transformations allow the correct and timely recomputation of incorrect attribute instances needed for additional transformations. In Sect. 6, by way of an example, the idea of the iteration of a transformation pass is discussed.

2. Basic Concepts

In the classical theory [17] an *attribute grammar* AG is based on a context-free grammar G, which is augmented with attributes and attribute evaluation rules. We recall some concepts and definitions that will be useful in the following parts.

The underlying context-free grammar $G=(V_N, V_T, P, S)$ is reduced. V_N and V_T denote the finite sets of nonterminal and terminal symbols respectively. We write V for $V_N \cup V_T$. P is the set of productions and S the start symbol.

We denote a production $p \in P$ as p: $X_{p0} \to X_{p1} X_{p2} \ldots X_{pn}$, where $n \geqq 0$, $X_{p0} \in V_N$ and $X_{pk} \in V$ for $1 \leqq k \leqq n$.

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

The set of all attributes will be denoted by A, i.e., $A=\bigcup_{X \in V} A(X)$. Attributes of different symbols are different. If necessary we will denote an attribute a **of** symbol X by $X \cdot a$. In examples we also write a **of** X rather than $X \cdot a$.

With each attribute a set of possible values is associated.

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 subsets of *defined occurrences* and *used occurrences* denoted by $DO(p)$ and $UO(p)$ respectively. These subsets are defined as follows:

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

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

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 $(a, p, k) \in \text{DO}(p)$, f is a total function and $(a_j, p, k_j) \in \text{UO}(p)$ 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 template we also need to concept of a "possibly incomplete" derivation tree where arbitrary symbols may label the root and the leaves. Apart form 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 there is a production p: $X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn}$, such that the node is labeled with X_{p0} and its n sons are labeled with X_{p1}, $X_{p2}, \ldots, X_{pn}$, respectively. We say that p is the production (*applied*) *at* that node.

Given a (possibly incomplete) 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* $N \cdot a$.

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

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

is associated with attribute instance $N_k \cdot a$ 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 $N_k \cdot a$ *depends on* $N_{k_i} \cdot a$ for $1 \leqq i \leqq m$.

For each (possibly incomplete) derivation tree T a *dependency graph* D_T can be defined by taking the attribute instances of T as its vertices. Arc $(N_i \cdot a, N_j \cdot b)$ is contained in the graph if and only if attribute instance $N_j \cdot b$ depends on attribute instance $N_i \cdot a$. The arcs specify a partial ordering of the attribute instances. The existence of arc $(N_i \cdot a, N_j \cdot b)$ indicates that attribute instance $N_i \cdot a$ must be computed before attribute instance $N_j \cdot b$. Later (in Sect. 4) we will add labels to the arcs of D_T.

A path in a dependency graph will be called a *dependency path*. For dependency paths we shall use the following notation: dp$[N_1 \cdot a_1, N_2 \cdot a_2, \ldots, N_n \cdot a_n]$

for $n>1$ stands for a path, composed of the arcs $(N_1 \cdot a_1, N_2 \cdot a_2)$, $(N_2 \cdot a_2, N_3 \cdot a_3)$, ..., $(N_{n-1} \cdot a_{n-1}, N_n \cdot a_n)$.

The task of an attribute evaluator is to compute the values of all attribute instances attached to the derivation tree, by executing the attribute evaluation instructions associated with these attribute instances. In general the order of evaluation is free, with the only restriction that an attribute evaluation instruction cannot be executed before its arguments are available. An attribute instance is *available* if its value is defined, otherwise it is *unavailable*. Initially all attribute instances attached to the derivation tree are unavailable, with the exception of the synthesized attribute instances associated with terminal symbols. The latter are assigned by the parser. The attribute evaluator has completed its task when all attribute instances are available. An *attributed derivation tree* is a derivation tree for which all attribute instances are available.

An overview of attribute evaluation methods is presented in [8].

3. Conditional Tree Transformations

In this section we define the concept of a conditional tree transformation rule and discuss the strategy of performing tree transformations during a single pass over a derivation tree, where a pass over a tree is defined to be a depth-first left-to-right or right-to-left traversal of the tree. In this paper we restrict ourselves to a transformation pass from left to right. The discussion of a right-to-left pass is analogous to the one of a left-to-right pass.

We start by defining the notion of an (unconditional) tree transformation rule consisting of two tree templates.

Definition 3.1. A *tree template* is a possibly incomplete derivation tree. Multiple occurrences of the same symbol as the label of a node should be distinguished by indices. Nonterminal symbols labeling the leaves are the *variables* of the tree template. So, in general, variables are of the form $X[i]$ where X is a nonterminal and i an index. □

Definition 3.2. 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. □

Definition 3.3. [7] A *tree transformation rule* is a pair (itt, ott) of tree templates, such that all variables occurring in ott also occur in itt; itt and ott are called the *input tree template* and the *output tree template* respectively. □

A tree transformation rule (itt, ott) is *applicable* to a subtree IT of a derivation tree $T1$ if itt *matches* the top of IT, i.e., if IT is an instance of itt. The fact that IT is an instance of itt establishes a relation between the variables of itt and subtrees of IT.

The *application* of tree transformation rule (itt, ott) consists of the creation of an instance OT of ott in which the relation between subtrees and variables 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 $T2$.

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

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

To guarantee that OT correctly fits in the surrounding tree $T2$ (i.e., that the production applied above OT preserves the syntax), it is necessary and sufficient to require that grammar symbol A labeling the root of itt may be replaced by grammar symbol B labeling the root of ott, at any occurrence of A in the right part of any production. However, for reasons of simplicity we impose an additional requirement on the transformation rules, namely that itt and ott have equally labeled roots. This is not a serious restriction. Generally a tree template is a superposition of productions, which means that it is always possible to extend itt and ott with an extra production so that their roots are labeled equally. Similarly, we require both the input template and the output template to consist of more than one node.

Now we decorate our tree templates with attributes and introduce the concept of a conditional tree transformation rule [11, 25] to transform attributed derivation trees.

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 case of a single occurrence of X in tt. We say that tree template tt has *attribute instance* $(a, tt, X[i])$ if $a \in A(X)$. $(a, tt, X[i])$ is an *inherited* instance if $a \in I(X)$, and a *synthesized* instance if $a \in S(X)$.

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

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

Definition 3.4. With respect to a tree template,

the *input attribute instances* are the inherited attribute instances of its root and the synthesized attribute instances of its leaves;

the *output attribute instances* are the synthesized attribute instances of its root and the inherited attribute instances of its leaves;

the *local attribute instances* are the attribute instances of its inner nodes. □

Observe that for each tree template the values of the input attribute instances completely determine the values of the local and the output attribute instances.

To shorten our explanations we will no longer discriminate a tree template from its corresponding area in the derivation tree. For this reason it is not always possible to make a distinction between the attribute instances of a tree template and the corresponding attribute instances of a derivation tree.

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

matching criterion. In general the applicability of a tree transformation rule may be further restricted by contextual information, collected and distributed by attributes. So, for attributed derivation trees, we extend the transformation rules by *enabling conditions* which are predicates on attribute instances of the input template. In this paper we allow the enabling conditions to be formulated in terms of input attribute instances only. This is a natural restriction since, as noted above, it is always possible to express the values of the local and the output instances in terms of the values of the input instances.

Next we focus on the correct 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 itt by the output template ott. No syntactical changes take place in the subtrees substituted for corresponding variables of itt and ott. Notice that also the production above the restructured subtree remains unchanged, since we required itt and ott to have equally labeled roots. So, we assume that the attribute instances of the subtrees substituted for the variables of itt and ott keep their values after a transformation. The same holds for the attribute instances of the tree part above itt and ott. Thus, we may now restrict ourselves to the evaluation of the attribute instances of ott. First of all we assume that the input attribute instances of the root and the variables of itt and ott keep their values. The same is assumed for the synthesized attribute instances of the terminal nodes (i.e., nodes labeled by a terminal symbol) of ott for which a corresponding node (i.e., a node with the same indexed label) exists in itt. Explicit evaluation rules are, however, needed for the synthesized attribute instances associated with terminal nodes of ott for which no corresponding node exists in itt. These attribute instances (normally set by the parser!) will be defined in terms of attribute instances of itt.

The local and the output attribute instances of ott are now uniquely determined (by the input attribute instances of ott and the ordinary attribute evaluation rules). However, to shortcut calling the evaluator (especially, in case ott is "large"), we may also specifiy them by special rules that define their values as a function of certain attribute instances of itt. In particular we could indicate that the values of certain corresponding attribute instances of itt and ott have to be transferred unchanged from itt to ott. To do this, we of course have to be sure that these rules compute the correct attribute values.

Consider the application of a tree transformation rule (itt, ott) which transforms a derivation tree $T1$ into a derivation tree $T2$. In general the values of some of the output attribute instances of ott in $T2$ will differ from the values of the corresponding output attribute instances of itt in $T1$. Let $N_1 \cdot a$ be an output attribute instance of ott whose new value differs from its old value. Then in $T2$ each attribute instance $N_2 \cdot b$, such that dependency graph D_{T2} includes a dependency path $\mathrm{dp}[N_1 \cdot a, \ldots, N_2 \cdot b]$, may have a wrong value. A tree transformation may even cause the values of the input attribute instances of ott to be incorrect (and hence the local too).

For further tree transformations we need the correct values of attribute instances to be used in enabling conditions. Hence, after the application of a tree transformation rule (itt, ott), it may be necessary to do some re-evaluations, not only in ott, but also elsewhere in $T2$.

A simple solution is to perform a complete re-evaluation of the whole of $T2$. However, a repeated computation (after each application of a transformation rule) of all the attribute instances attached to the derivation tree is time consuming and cannot be accepted as suitable solution to the problem.

To minimize work, the re-evaluation process could be confined – as much as possible – to those attribute instances whose values became incorrect by the transformation of the derivation tree. This approach is followed by Möncke c.s. in [19] for OPTRAN [11], a language and a system designed to describe and to perform tree transformations, by Alblas in [3], and also by Reps c.s. in [6, 23, 24], Yeh in [26] and Engelfriet in [8] for syntax directed editing, where the decisions to re-evaluate attributes are made on the fly, i.e., at compile time.

A different approach will be investigated in this paper. Our tree transformation and re-evaluation strategy has two characteristics.

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

2. The re-evaluation process after each tree transformation is confined to attribute instances which are met during the continuation of the transformation pass and whose values are not known to be correct (because they depend on attribute instances whose values were possibly changed by a tree transformation). The transformation pass will never be interrupted in order to make extra tree traversals for re-evaluation purposes. The consequence of this strategy is that the correct re-evaluation (if necessary at all) of attribute instances that cannot be recomputed (correctly) during the continuation of the transformation pass has to be delayed until the transformation pass has been finished.

A prerequisite for the application of this strategy to a given attribute grammar and a given set of tree transformation rules is the necessity to know in advance (i.e., at compiler construction time rather than at compile time) whether it is "always safe" to perform tree transformations and re-evaluations during a single pass. By "always safe" we mean that for any derivation tree each attribute instance whose value might become incorrect as a result of a tree transformation, but whose value is needed directly or indirectly for the enabling conditions of additional tree transformations, can be recomputed correctly and in time during the continuation of the pass.

This means in particular, that it must be clearly indicated for which output attribute instances of the output template of a tree transformation rule the value may change as a result of the tree transformation.

We adopt the following strategy to make this visible: The values of the local and the output attribute instances of the output template are assumed to be taken over unchanged from the input template unless an explicit re-evaluation rule is specified. We require the re-evaluation rules to be expressed in terms of the input attribute instances of the input template only, since they completely determine the values of the local and the output attribute instances of the output template. So, the local and the output attribute instances of the output template may be defined in two distinct ways:

1) explicitly, by re-evaluation rules which define their values as a function of input attribute instances of the input template;

2) implicitly, by leaving out their re-evaluation rules, in which case their values are taken over unchanged from the corresponding attribute instances of the input template.

Since the evaluation rules for attribute instances of terminal nodes of the output template also satisfy these two points, we will also call them re-evaluation rules. This leads to the following formal definition of an attribute re-evaluation rule.

Definition 3.5. Let (itt, ott) be a tree transformation rule, and let (a, ott, Y) be an attribute instance of ott that is *not* an input attribute instance of the root or the variables of ott. An *attribute re-evaluation rule* for (a, ott, Y) is of 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 total function and (a_j, itt, Y_j) is an input attribute instance of itt, for $1 \leq j \leq m$. □

Having decided to perform tree transformations during a pass over the derivation tree, we also have to indicate when a transformation rule has to be applied during such a pass.

Consider a subtree which may be restructured by the application of a tree transformation rule. During the transformation pass 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 done 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 the downward move or during the upward move.

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

Definition 3.6. A *conditional tree transformation rule* is a quintuple 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 tree templates: the *input* and the *output template* respectively. itt and ott have equally labeled roots and consist of more than one node. All variables occurring in ott also occur in itt;

– cond is the *enabling condition*, a predicate on input attribute instances of itt;

– eval is the set of *attribute re-evaluation rules* for attribute instances of ott. □

Definition 3.7. A conditional tree transformation rule tr: (dir, itt, ott, cond, eval) is *consistent* if the following two statements hold:

(1) Each attribute of ott for which there is no attribute re-evaluation rule in eval, has a corresponding attribute instance in itt.

(2) Whenever the attribute instances of itt are given correct values (in the sense that these values satisfy the attribute evaluation rules), then the attribute

instances of ott obtain correct values (in the same sense) by the following computation:

– the value of each attribute instance of ott for which there is an attribute re-evaluation rule, is computed according to that rule;
– the value of each attribute instance of ott for which there is no such rule is copied from the corresponding attribute instance of itt. □

The consistency of a conditional tree transformation rule is the responsibility of the writer of these rules, i.e., it is not checked at compiler construction time (that would in fact be impossible). Thus, throughout this paper, we will assume that all conditional tree transformation rules are consistent.

Note that for each output or local attribute instance of ott the attribute re-evaluation rule (if present in eval) is uniquely determined by the consistency requirement: it can be obtained by first expressing the attribute instance in terms of the input instances of ott, using the attribute evaluation rules of the grammar, and then replacing each attribute instance of a terminal for which there is an attribute re-evaluation rule in eval by the right-hand side of that rule, and replacing every other input attribute instance of ott by the corresponding input attribute instance of itt.

Making a pass over an attributed derivation tree a conditional tree transformation rule tr: (dir, itt, ott, cond, eval) is *applicable* to a subtree IT, after a downward or an upward move to the root of IT, if the following conditions are satisfied:

1) the direction of the move corresponds with the value of dir;
2) itt matches the top of IT;
3) the evaluation of cond yields *true*.

The *application* of transformation rule tr includes the 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. Moreover, values of attribute instances of ott are computed according to the re-evaluation rules specified by eval. The values of attribute instances of ott for which no re-evaluation rule is specified, are copied from itt. After the application of tr the pass continues by entering or leaving OT.

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 $=$ *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

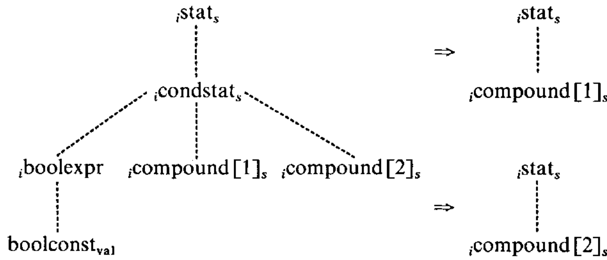

Fig. 1. Replacement of a conditional statement by one of its components

We illustrate our explanations with a small example, analogous to a more comprehensive example concerning data flow analysis, constant propagation and dead code elimination in Sect. 6.

For the specification of tree templates we use the following linear notation for trees: within the bracked pair (⟨, ⟩) the root is followed by its sequence of subtrees. Comma symbols are used 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 grammar symbols 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 instances (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 rules

trans: **transform up** ⟨stat, ⟨condstat, ⟨boolexpr, boolconst⟩, compound [1], compound[2]⟩⟩
  cond val **of** boolconst = *true*
    into ⟨stat, compound[1]⟩
      eval s **of** stat := s **of** compound[1]
  cond val **of** boolconst = *false*
    into ⟨stat, compound[2]⟩
      eval s **of** stat := s **of** compound[2]
end

describe the replacement of a conditional statement, in case of a constant boolean expression, by its then part or its else part. See Fig. 1, where inherited attribute instances have been written to the left and synthesized attribute instances to the right of their associated grammar symbol.

Observe that no re-evaluation rules have been included for i **of** compound[1] and i **of** compound[2]. Apparently their values can be copied from the corresponding attribute instances in itt. □

4. Dependencies Between Attributes Involved in Successive Transformations

In order to know whether and how a transformation of a derivation tree may influence additional transformations we investigate for each pair of successively

applicable tree transformation rules the possible dependencies between output attribute instances of the output template of the first one and input attribute instances of the input template of the second one.

This investigation may be restricted to the output attribute instances whose values may change as a result of the first transformation and the input attribute instances whose values have to be correct because they are used as arguments for the enabling condition and the re-evaluation rules of the second transformation. In fact it is sufficient to require only the correctness of those arguments of re-evaluation rules that may influence additional transformations, but this will lead to a complicated safety test (i.e., the test whether the recomputation of possibly incorrect but necessary attribute instances can be done correctly and in time during the transformation pass).

For this reason we introduce for each transformation rule tr the sets COR(tr) of attribute instances that have to be correct and EVAL(tr) of attribute instances for which a re-evaluation rule is specified.

Definition 4.1. For each tree transformation rule tr: (dir, itt, ott, cond, eval),
– COR(tr) is the set of all input attribute instances of itt, used as arguments for cond or re-evaluation rules in eval,
– EVAL(tr) is the set of all output attribute instances of ott, for which eval contains a re-evaluation rule. □

For every pair of tree transformation rules we will exclude any overlap of the output template of the first one and the input template of the second one (when the second is applied after the first). By overlap of two tree templates in a derivation tree we mean that the templates have at least two tree nodes in common. Overlap does not include the case where the tree templates just touch, i.e., have one tree node in common. This occurs when a leaf of the first template is the root of the second template or the other way round.

Let q: (dir_q, itt_q, ott_q, cond_q, eval_q) and r: (dir_r, itt_r, ott_r, cond_r, eval_r) be conditional tree transformation rules. Consider the situation where no transformation is performed between the (non-overlapping) applications of q and r.

Let Rott_q and Ritt_r label the roots of subtrees OT_q and IT_r, respectively, where OT_q is an instance of ott_q and IT_r is an instance of itt_r. For a tree transformation pass from left to right we say that itt_r *is found after* ott_q in a derivation tree T, if ott_q and itt_r do not overlap in T, and one of the following three conditions is satisfied.

a) Rott_q and Ritt_r are *cousins*. This means that in T a production p: $X_{p0} \to X_{p1} \ldots X_{pj} \ldots X_{pk} \ldots X_{pn}$ has been applied such that $\text{Rott}_q = X_{pj}$ or Rott_q is a descendant of X_{pj}, and $\text{Ritt}_r = X_{pk}$ or Ritt_r is a descendant of X_{pk}.
   Both dir_q and dir_r may be *up* or *down*.
b) Ritt_r is a *descendant* of Rott_q. More precisely, Ritt_r is a leaf or a descendant of a leaf of ott_q.
   dir_q must be *down* and dir_r may be *up* or *down*.
c) Ritt_r is an *ancestor* of Rott_q. More precisely, Rott_q is a leaf or a descendant of a leaf of itt_r.
   dir_q may be *up* or *down* and dir_r must be *up*.

For a transformation pass from left to right the three different cases, where itt_r is found after ott_q, have been pictured in Fig. 2.

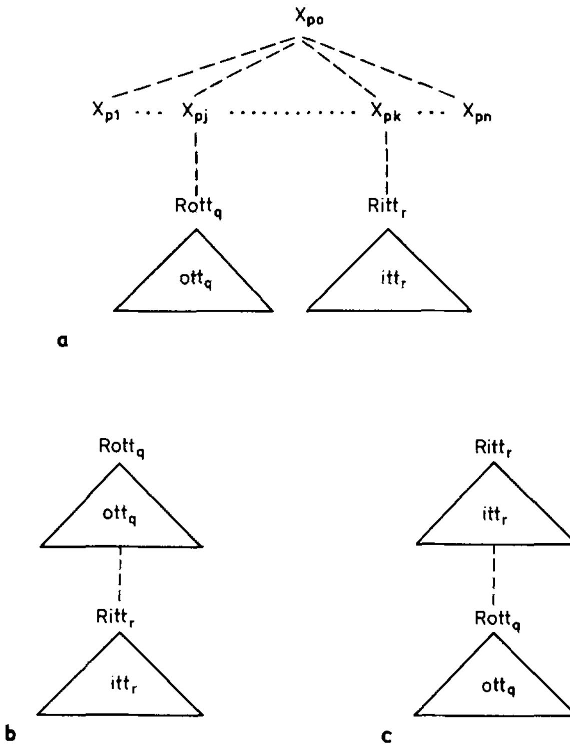

Fig. 2a–c. Relative positions of tree templates. **a** Rott_q and Ritt_r are cousins. **b** Ritt_r is a descendant of Rott_q; **c** Ritt_r is an ancestor of Rott_q

For each derivation tree T including the output template ott_q of a tree transformation rule q and the input template itt_r of an additional tree transformation rule r, we can use its dependency graph D_T to express the dependencies between attribute instances of EVAL(q) and attribute instances of COR(r). To simplify and to shorten our explanations, we will not always make a clear distinction between a derivation tree T and its dependency graph D_T. Sometimes we will talk about a (dependency) path in T if we actually mean a dependency path in D_T.

Let a pair of tree transformation rules q: (dir_q, itt_q, ott_q, cond_q, eval_q) and r: (dir_r, itt_r, ott_r, cond_r, eval_r) be given. We say that r can be applied *safely after* q during a transformation pass if for any derivation tree T, including an instance of ott_q and an instance of itt_r such that itt_r is found after ott_q, each attribute instance whose value may be incorrect as a result of the application of q but whose value is needed as an argument for the enabling condition or the re-evaluation rules of r, can be recomputed correctly after the application of q and before the application of r (note that these applications are assumed to be non-overlapping).

Let us state more precisely what we mean by this. Let T be such a derivation tree. T includes both a subtree OT_q which is an instance of ott_q and a subtree

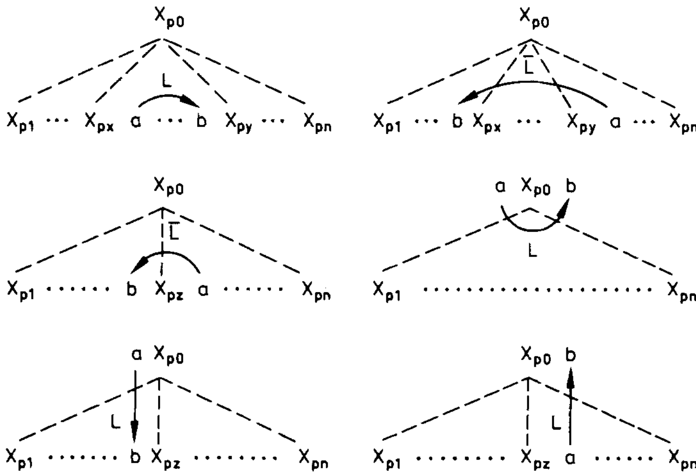

Fig. 3. Attribute dependencies of a production

IT_r which is an instance of itt_r. We say that an attribute instance *may be incorrect* as a result of the application of q, if there is a dependency path in T from an element of EVAL(q) to the attribute instance. Thus, in T, the application of transformation rule r is safely possible after the application of transformation rule q if for every pair of attribute instances $(a, \mathrm{ott}_q, X) \in \mathrm{EVAL}(q)$ and $(b, \mathrm{itt}_r, Y) \in \mathrm{COR}(r)$, the following holds: for each dependency path in T from (a, ott_q, X) to (b, itt_r, Y) it must be possible to recompute the attribute instances on the path correctly during the transformation pass after the application of q and before the application of r. Notice that such a dependency path not necessarily follows the path of the pass. To explain this, we consider the dependencies induced by the evaluation rules of a production.

With each production p: $X_{p0} \to X_{p1} X_{p2} \ldots X_{pn}$, we associate a labeled dependency graph DG_p as follows. The vertices of DG_p are the attribute occurrences of production p. For $0 \leq j, k \leq n$ there is a directed arc from (a, p, j) to (b, p, k) in DG_p if and only if (b, p, k) depends on (a, p, j). Up to now this is the usual definition [17]. We now discuss the labeling of the arcs.

For a left-to-right pass the visiting order of the attribute occurrences of production p is as follows: occurrences of inherited attributes of X_{p0}, occurrences of inherited attributes of X_{p1}, occurrences of synthesized attributes of $X_{p1}, \ldots,$ occurrences of inherited attributes of X_{pn}, occurrences of synthesized attributes of X_{pn}, occurrences of synthesized attributes of X_{p0}.

In accordance with this to each arc a label is assigned in the following way. An arc from (a, p, j) to (b, p, k) is labeled L if and only if the following condition is satisfied: if $1 \leq j, k \leq n$ then $j < k$; otherwise the arc is labeled $\bar{L}$. Figure 3 shows the six basic dependencies to be distinguished.

Observe that the existence of an arc from (a, p, j) to (b, p, k) with label L implies that (b, p, k) depends on (a, p, j) and that during a left-to-right pass (a, p, j) will be found before (b, p, k). An arc labeled $\bar{L}$ means that (b, p, k)

depends on (a, p, j) but that during a left-to-right pass (a, p, j) will not be found before (b, p, k).

Consider a derivation tree T and its associated dependency graph D_T. Let N_0 be a node of T, p the production applied at N_0 and $N_1, \ldots, N_n$ the sons of N_0 from left to right respectively. If DG_p includes an arc from (a, p, j) to (b, p, k) then D_T includes an arc $(N_j \cdot a, N_k \cdot b)$. To arc $(N_j \cdot a, N_k \cdot b)$ in D_T we assign the same label as to the arc from (a, p, j) to (b, p, k) in DG_p. In this manner we attach a label L or $\bar{L}$ to every arc in D_T.

Now we come back to the recomputation of attribute instances in a derivation tree T after the application of a transformation rule q and before the application of an additional transformation rule r. If transformation rule r can be applied safely after the application of transformation rule q then for every pair of attribute instances $(a, \text{ott}_q, X) \in \text{EVAL}(q)$ and $(b, \text{itt}_r, Y) \in \text{COR}(r)$ the following holds: each arc of each dependency path from (a, ott_q, X) to (b, itt_r, Y) must have label L. In fact, an arc labeled $\bar{L}$ prevents the correct recomputation of (b, itt_r, Y) during a single left-to-right pass.

Another essential point is the position of X and Y within ott_q and itt_r respectively. If $\text{dir}_q = up$ and X labels a leaf of ott_q then every dependency path from (a, ott_q, X) to (b, itt_r, Y) runs via the subtree whose root has label X. Hence, an extra traversal of this subtree is needed for the recomputation of (b, itt_r, Y). This conflicts with the upward move of the tree walk just after the first transformation. A similar argument holds for the case where $\text{dir}_r = down$ and Y labels a leaf of itt_r. Then every dependency path from (a, ott_q, X) to (b, itt_r, Y) passes through the subtree whose root has label Y. This means that a visit to this subtree has to be made before the application of r, which conflicts with the downward move of the tree walk just before the second transformation.

Using these observations it is not difficult to prove the following theorem.

Theorem 4.1. *Let* AG *be an attribute grammar. Let* q: $(\text{dir}_q, \text{itt}_q, \text{ott}_q, \text{cond}_q, \text{eval}_q)$ *and* r: $(\text{dir}_r, \text{itt}_r, \text{ott}_r, \text{cond}_r, \text{eval}_r)$ *be conditional tree transformation rules.*

Transformation rule r can be applied safely after *the application of transformation rule q during a pass from left to right if for every pair of attribute instances* $(a, \text{ott}_q, X) \in \text{EVAL}(q)$ *and* $(b, \text{itt}_r, Y) \in \text{COR}(r)$, *the following holds*: *for each derivation tree T including subtrees* OT_q *and* IT_r, *which are instances of* ott_q *and* itt_r *respectively, such that* itt_r *is found after* ott_q, *if a dependency path exists from* (a, ott_q, X) *to* (b, itt_r, Y), *then*:

(a) *no such dependency path includes an* arc *labeled* $\bar{L}$.

(b) *if* $\text{dir}_q = \text{up}$ *then X labels the root of* ott_q.

(c) *if* $\text{dir}_r = \text{down}$ *then Y labels the root of* itt_r. □

A set TR of tree transformation rules will be called *pairwise safe* if for all q and r in TR, r can be applied safely after q during a left-to-right pass.

Now, given such a set TR, the question arises whether it is also possible to accomplish a correct recomputation of necessary attribute instances after the (non-overlapping) application of any sequence of tree transformations.

Let TR be a set of tree transformation rules. We say that TR is *safe* with respect to a left-to-right pass if for each r: $(\text{dir}_r, \text{itt}_r, \text{ott}_r, \text{cond}_r, \text{eval}_r) \in \text{TR}$ and any sequence $q1, \ldots, qn$, where qk: $(\text{dir}_{qk}, \text{itt}_{qk}, \text{ott}_{qk}, \text{cond}_{qk},$

$\text{eval}_{qk}) \in \text{TR}(1 \leq k \leq n)$ the following holds: for any derivation tree T', generated from an original derivation tree T by the successive (non-overlapping) application of $q1, \ldots, qn$ during a transformation pass and such that itt_r is found after ott_{qn} in T', each attribute instance whose value may be incorrect as a result of the application of any tree transformation rule $qk(1 \leq k \leq n)$ but whose value is needed as an argument for the enabling condition or the re-evaluation rules of r, can be recomputed correctly before the application of r.

Theorem 4.2. *Let* TR *be a set of tree transformation rules defined for an attribute grammar* AG. *With respect to a left-to-right pass,* TR *is safe if and only is* TR *is pairwise safe.*

Proof. ($\Rightarrow$) TR is safe implies TR is pairwise safe.

($\Leftarrow$) We have to prove that each r: (dir_r, itt_r, ott_r, cond_r, eval_r)$\in$TR can be applied safely after the (non-overlapping) application of any sequence of tree transformation rules from TR. We prove this part of the theorem by induction on the number of preceding tree transformations.

I. Induction basis.

From TR is pairwise safe it follows that r can be applied safely after the application of any single transformation rule from TR.

II. Induction step.

Induction hypothesis: the statement holds for any preceding sequence of length n. We have to prove that it holds for all sequences of length $n+1$.

Consider a sequence $q1, \ldots, qn, s$ to be applied during a pass before the application of r. Let T0 be the original derivation tree before the application of any tree transformation rule. Let $T1$ be the derivation tree generated from T0 by the consecutive application of transformation rules $q1, \ldots, qn$ and let $T2$ be the derivation tree after the application of s on $T1$. Let T0′ be the derivation tree obtained from T0 by the application of s. (Due to the non-overlapping application of transformation rules) $T2$ can also be generated from T0′ by the application of $q1, \ldots, qn$.

We will use the following terminology: when we say that "a path can be followed in $T2$", we mean that all attribute instances of the path are visited in the order they have on the path, during the transformation pass in which $q1, \ldots, qn, s, r$ are applied to T0 (moreover, the value of the first attribute instance of the path should be correct). Similarly for a path in $T1$ when $q1, \ldots, qn, r$ are applied to T0.

If all dependency paths in $T2$ of the following three types can be followed in $T2$, then the correct value of the elements of COR(r) in $T2$ can be recomputed.

(1) a path from an attribute instance $(a, \text{ott}_s, A) \in \text{EVAL}(s)$ to an attribute instance $(b, \text{itt}_r, B) \in \text{COR}(r)$;

(2) a path from an attribute instance $(a, \text{ott}_{qk}, A) \in \text{EVAL}(qk)$ $(1 \leq k \leq n)$ to an attribute instance $(b, \text{itt}_r, B) \in \text{COR}(r)$, not running through ott_s:

(3) a path from an attribute instance $(a, \text{ott}_{qk}, A) \in \text{EVAL}(qk)$ $(1 \leq k \leq n)$ to an attribute instance $(b, \text{itt}_r, B) \in \text{COR}(r)$, running through ott_s but not through any element of EVAL(s).

Consider a path of the first type: from an attribute instance $(a, \text{ott}_s, A) \in$ EVAL(s) to an attribute instance $(b, \text{itt}_r, B) \in \text{COR}(r)$. There is a re-evaluation

rule for (a, ott_s, A) in eval_s. The arguments of this rule are in COR(s). Hence, by the induction hypothesis for $T1$ (for the sequence $q1, \ldots, qn, s$ applied to T0), the correct values of these arguments in $T1$ can be recomputed. Thus, application of the re-evaluation rule gives the same value in $T2$ to (a, ott_s, A) as it would get when $T1$ was correctly attributed. Now the safety of r after s implies that the path can be followed in $T2$ after the application of s.

A path in $T2$ of the second type will also exist in $T1$. From the induction hypothesis for $T1$ (in this case, for the sequence $q1, \ldots, qn, r$ applied to T0) it follows that such a path can be followed in $T1$ (i.e., when transformation rule s is not applied). From the fact that the path does not enter ott_s it follows that it is also possible to follow the same path in $T2$.

Finally consider a path of the third type: from an attribute instance $(a, \text{ott}_{qk}, A) \in \text{EVAL}(qk)$ to an attribute instance $(b, \text{itt}_r, B) \in \text{COR}(r)$, running through ott_s but not through any element of EVAL(s). Thus, all output attribute instances of ott_s that lie on the path, are not in EVAL(s). Consider a piece of the path inside ott_s, leading from an input attribute instance (c, ott_s, C) to an output attribute instance (d, ott_s, D) of ott_s. Clearly, there are corresponding attribute instances (c, itt_s, C) and (d, itt_s, D) of itt_s. It now follows from the consistency of s and the fact that $(d, \text{ott}_s, D) \notin \text{EVAL}(s)$ that there is also a (piece of) path inside itt_s from (c, itt_s, C) to (d, itt_s, D). Hence, there is a corresponding path from (a, ott_q, A) to (b, itt_r, B) in $T1$, running through itt_s in the same way as the original path runs through ott_s in $T2$. Hence, by the induction hypothesis for $T1$ (for the sequence $q1, \ldots, qn, r$ applied to T0), the path can be followed in $T1$ up to the application of s (i.e., for $\text{dir}_s = \textit{down}$: up to the first input attribute instance of itt_s, and for $\text{dir}_s = \textit{up}$: up to the last output attribute instance of itt_s). The rest of the path can be followed in $T2$, by the induction hypothesis for $T2$ (for the sequence $q1, \ldots, qn, r$ applied to T0′). □

Remark. The fact that COR(tr) includes the arguments for the re-evaluation rules in eval_{tr}, guarantees the correctness of the new starting points of data flow paths (i.e., the correctness of the synthesized attribute instances of the newly created terminal symbols of ott_{tr}). In fact it is sufficient to require only the correctness of those attribute instances of terminal nodes that may influence additional transformations. But this will lead to a complicated safety test in Section 5.

For attribute instances in EVAL(tr), associated with nonterminal nodes of ott_{tr}, it is generally not necessary to include their arguments in COR(tr), except in case of a conditional tree transformation rule tr to be applied during a bottom up move, and of which ott_{tr} includes a data flow path for which there is no corresponding path in itt_{tr}. Instead of requiring the correctness of the arguments in question one could also forbid a tree transformation rule to add data flow paths.

In this paper we ignore these problems and simply require COR(tr) to include the arguments for all attribute instances in EVAL(tr).

5. An Algorithm to Test the Safety of Successive Transformations

Let AG be an attribute grammar and TR a set of tree transformation rules defined for AG. We will show in this section that there is an algorithm that decides whether or not TR is safe for AG. As proved in Theorem 4.2 the (non-overlapping) application of any sequence of tree transformation rules from TR during a single left-to-right pass is safe if and only if TR is pairwise safe. Thus it suffices to show that it can be decided whether or not one tree transformation rule can be applied safely after another.

In this section for every derivation tree T and for every pair of tree transformation rules q: (dir_q, itt_q, ott_q, cond_q, eval_q) and r: (dir_r, itt_r, ott_r, cond_r, eval_r) such that itt_r is found after ott_q in T, we will show how to compute the dependency relations between attribute instances in EVAL(q) and attribute instances in COR(r) in order to find out, using Theorem 4.1 as a criterion, whether r can be applied safely after q.

5.1 A First Investigation: Necessary Graphs

Let Rott_q and Ritt_r label the roots of ott_q and itt_r respectively. In Section 4 we distinguished the following cases for the relative positions of Rott_q and Ritt_r.

a) Rott_q and Ritt_r are cousins,

b) Ritt_r is a descendant of Rott_q,

c) Rott_q is a descendant of Ritt_r.

Case a. See Fig. 2a. Let T be a derivation tree including tree templates ott_q and itt_r, where itt_r is found after ott_q and such that Rott_q and Ritt_r are cousins. Let p: $X_{po} \to X_{p1} \dots X_{pj} \dots X_{pk} \dots X_{pn}$ be the production applied in T such that $\mathrm{Rott}_q = X_{pj}$ or Rott_q is a descendant of X_{pj} and $\mathrm{Ritt}_r = X_{pk}$ or Ritt_r is a descendant of X_{pk}. To construct the dependency graph which expresses the dependency relations between attribute instances in EVAL(q) and attribute instances in COR(r) we need the following dependency relations in the following parts of T.

a.1) dependencies between attribute instances $A(X_{pj})$ and $A(X_{pk})$ in the tree $T1$ obtained from T by deleting its subtrees with root X_{pj} and X_{pk};

a.2) dependencies between attribute instances $A(X_{pj})$ and $A(\mathrm{Rott}_q)$ in the tree $T2$ obtained from T by isolating its subtree with root X_{pj} and deleting its subtree with root Rott_q;

a.3) dependencies between attribute instances $A(X_{pk})$ and $A(\mathrm{Ritt}_r)$ in the tree $T3$ obtained from T by isolating its subtree with root X_{pk} and deleting its subtree with root Ritt_r;

a.4) dependencies between attribute instances of ott_q in the tree $T4$ obtained from T by isolating its subtree with root Rott_q;

a.5) dependencies between attribute instances of itt_r in the tree $T5$ obtained from T by isolating its subtree with root Ritt_r.

Observe that by "deleting a subtree" is meant: "deleting a subtree, except its root".

In the following we do not restrict our computations to a single derivation tree but we consider every possible derivation tree which includes tree templates

ott_q and itt_r such that itt_r is found after ott_q and such that their roots are cousins.

In Sects. 5.2, 5.3 and 5.4 we will develop algorithms to compute the above-mentioned dependency relations. In this section introductory remarks are made with respect to each of the parts (a.1), ..., (a.5).

Part a.1: For every pair of grammar symbols X and Y and for every production $p \in P$ with X to the left of Y in the right-hand side of p we are interested in all the graphs expressing dependency patterns between attribute occurrences of X and Y of production p applied in derivation trees whose subtrees rooted in X and Y have been deleted. This set of graphs where X and Y are "brothers", for X the "left brother", and Y the "right brother", will be called BROTHER-SET(X, Y).

Consider a production p: $X_{p0} \to X_{p1} \ldots X_{pj} \ldots X_{pk} \ldots X_{pn}$ such that $X = X_{pj}$ and $Y = X_{pk}$ as shown in Fig. 4 where attribute occurrences have been printed as large dots. To compute the above-mentioned dependencies between attribute occurrences of X_{pj} and X_{pk} we first of all need the dependency graph DG_p of production p. We also have to take into account for every possible application of production p the dependencies induced by its context. For X_{p0} we need the dependency graphs (with set of vertices $A(X_{p0})$) showing how its inherited attributes may depend on its synthesized attributes, i.e., the different s-to-i attribute dependency patterns at X_{p0} of trees whose subtree rooted in X_{p0} has been deleted. This set of graphs will be called SI-SET(X_{p0}). For every $X_{pi} (1 \leqq i \leqq n, i \neq j, k)$ we need the dependency graphs (with set of vertices $A(X_{pi})$) showing how its synthesized attributes may depend on its inherited attributes, i.e., the different i-to-s attribute dependency patterns at X_{pi} of subtrees with root X_{pi}. This set of graphs will be called IS-SET(X_{pi}). The labeling of the arcs of these graphs will be discussed later.

In Fig. 4 attribute dependencies induced by the evaluation rules of production p have been pictured by arcs with solid lines. Attribute dependencies from SI-SET(X_{p0}) and IS-SET(X_{pi}) $(1 \leqq i \leqq n, i \neq j, k)$ have been pictured by arcs with dashed lines.

Having available the sets IS-SET(Z) for every $Z \in V$ and SI-SET(Z) for every $Z \in V_N$, an arbitrary "brother-graph" of the set BROTHER-SET(X, Y) is constructed as follows. Choose a production: p: $X_{p0} \to X_{p1} \ldots X_{pj} \ldots X_{pk} \ldots X_{pn}$ such that $X = X_{pj}$ and $Y = X_{pk}$. Choose an "si-graph" from SI-SET(X_{p0}) and an "is-graph" from each IS-SET(X_{pi}) $(1 \leqq i \leqq n, i \neq j, k)$. With the superposition of these graphs (as pictured in Fig. 4) a brother-graph can be associated as

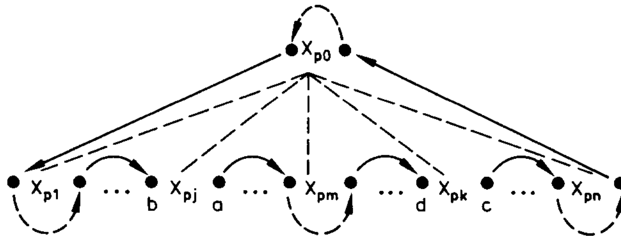

Fig. 4. Construction of a brother graph

follows. The vertices of the brother-graph are the attributes of $A(X)$ and $A(Y)$. For each pair of attributes $(X\cdot a,\ X\cdot b)$, $(X\cdot a,\ Y\cdot d)$, $(Y\cdot c,\ Y\cdot d)$ and $(Y\cdot c,\ X\cdot b)$, where $X\cdot a$ and $Y\cdot c$ are synthesized (used) attributes and $X\cdot b$ and $Y\cdot d$ are inherited (defined) attributes, an arc is included in the brother-graph if and only if there is an oriented path between the associated vertices in the graph constructed from DG_p and the selected *si*-graph and *is*-graphs.

For the graphs in BROTHER-SET(X, X), i.e., in the event of brothers of the same name, two instances of every attribute $a \in A(X)$ are needed. In this case subscripts "left" and "right" might be used to distinguish the left and the right instance of every attribute $a \in A(X)$ in the brother graphs concerned. The vertices of the graphs in BROTHER-SET$(X_{\text{left}}, X_{\text{right}})$ are the instances from $A(X_{\text{left}})$ and $A(X_{\text{right}})$. For the inclusion of the arcs we consider all productions p: $X_{p0} \to X_{p1} \ldots X_{pj} \ldots X_{pk} \ldots X_{pn}$, such that $X = X_{pj} = X_{pk}$. Furthermore we associate X_{left} and X_{right} with X_{pj} and X_{pk} respectively.

Parts a·2 and a·3: For every pair of grammar symbols $X \in V_N$, $Y \in V$ and for every derivation tree $T2$ with root X and whose subtree rooted in Y has been deleted we need the graph expressing the dependency relations between attribute instances of X and Y in $T2$. The set of all graphs where X and Y have such an "ancestor-descendant" relation will be called ANC-DESC-SET(X, Y).

These ancestor-descendant dependency relations are found by considering chains of productions as pictured in Fig. 5. Figure 6 shows a chain which consists of a single production. In both figures solid arcs indicate dependencies induced by the evaluation rules of the productions involved in the chain. Dashed arcs indicate dependencies from *is*-graphs.

To begin with we consider the case where X and Y form part of the same production. For every pair of grammar symbols $X \in V_N$, $Y \in V$ and for every production $p \in P$ such that X is the left-hand side and Y occurs in the right-hand side of p we are interested in all the graphs expressing dependencies between attribute occurrences of X and Y of production p applied in derivation trees with root X (i.e., p is the production at the root of the tree) and whose subtree rooted in Y has been deleted. This set of graphs where X and Y have a "father-son" relation for X the "father" and Y the "son" will be called FATHER-SON-SET(X, Y).

To construct an arbitrary "father-son" graph of the set FATHER-SON-SET(X, Y) we choose a production p: $X_{p0} \to X_{p1} \ldots X_{pk} \ldots X_{pn}$ such that X

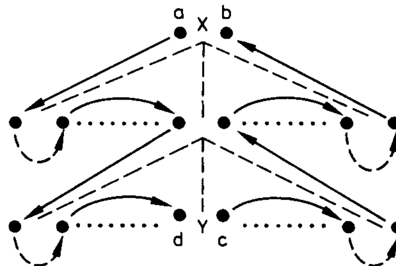

Fig. 5. Construction of an ancestor-descendant graph

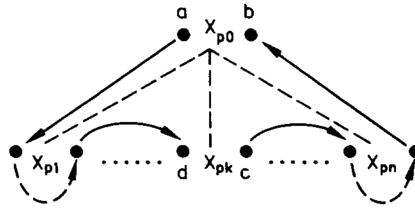

Fig. 6. Construction of a father-son graph

$=X_{p0}$ and $Y=X_{pk}$. Choose an is-graph from every set IS-SET(X_{pi}) $(1 \leqq i \leqq n,$ $i \neq k)$. With the superposition of these graphs (as pictured in Fig. 6) a father-son graph can be constructed as follows. The vertices are the attributes of $A(X)$ and $A(Y)$. For each pair of attributes $(X \cdot a,\ X \cdot b)$, $(X \cdot a,\ Y \cdot d)$, $(Y \cdot c,\ Y \cdot d)$ and $(Y \cdot c,\ X \cdot b)$, where $X \cdot a$ and $Y \cdot d$ are inherited attributes and $X \cdot b$ and $Y \cdot c$ are synthesized attributes, an arc is included in the father-son graph if and only if there is an oriented path between the associated vertices in the graph constructed from DG_p and the selected is-graphs.

For the graphs in FATHER-SON-SET(X, X) two instances of every attribute $a \in A(X)$ are needed, just as for BROTHER-SET(X, X).

Having available the sets FATHER-SON-SET(W, Z) for every pair of grammar symbols $W \in V_N$ and $Z \in V$, the construction of an arbitrary "ancestor- descendant" graph of the set ANC-DESC-SET(X, Y) proceeds as follows. Y is a descendant of X implies that a sequence $Z_0, Z_1, \ldots, Z_m$ of grammar symbols exists such that $Z_0 = X$ and $Z_m = Y$. From every set FATHER-SON-SET$(Z_i,$ $Z_{i+1})$ $(0 \leqq i \leqq m-1)$ we choose a father-son graph. The superposition of the selected father-son graphs in the order $Z_0, Z_1, \ldots, Z_m$ pictures dependency paths expressing all the dependencies between attributes of $A(X)$ and $A(Y)$ for a certain derivation tree with root X and whose subtree rooted in Y has been deleted. With the selected chain of father-son graphs an ancestor-descendant graph can be associated as follows. The vertices are the attributes of $A(X)$ and $A(Y)$. For each pair of attributes $(X \cdot a,\ X \cdot b)$, $(X \cdot a,\ Y \cdot d)$, $(Y \cdot c,\ Y \cdot d)$ and $(Y \cdot c,\ X \cdot b)$, where $X \cdot a$ and $Y \cdot d$ are inherited attributes and $X \cdot b$ and $Y \cdot c$ are synthesized attributes, an arc is included in the ancestor-descendant graph if and only if there is an oriented path between the associated vertices in the selected chain of father-son graphs.

For the graphs in ANC-DESC-SET(X, X) two instances of every attribute $a \in A(X)$ are needed, distinguished as $X_{\text{anc}} \cdot a$ and $X_{\text{desc}} \cdot a$.

Consider again Fig. 2a. A graph of the set ANC-DESC-SET$(X_{pj},\ \text{Rott}_q)$ has to be applied for Rott_q a descendant of X_{pj}. For $\text{Rott}_q = X_{pj}$ no such graph is needed. It is however not necessary to consider the situation, where Rott_q $= X_{pj}$, as a special case if we include the identity graph ident$(X_{\text{anc}}, X_{\text{desc}})$ in the set ANC-DESC-SET(X, X), for every grammar symbol X (including $X \in V_T$). For every inherited attribute $i \in I(X)$ and for every synthesized attribute $s \in S(X)$ ident$(X_{\text{anc}}, X_{\text{desc}})$ contains the arcs $(X_{\text{anc}} \cdot i,\ X_{\text{desc}} \cdot i)$ and $(X_{\text{desc}} \cdot s,\ X_{\text{anc}} \cdot s)$, respectively.

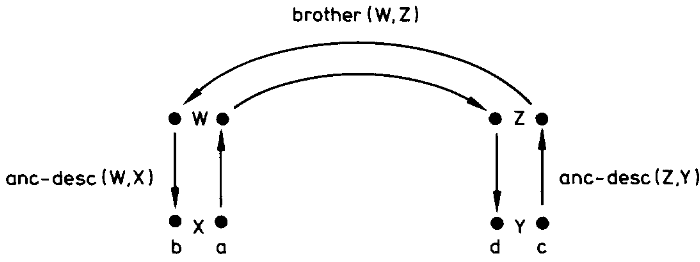

Fig. 7. Construction of a cousin graph

Parts a.1, a.2 and a.3: In case of a cousin relation between grammar symbols X and Y, we are interested in all the graphs expressing dependency patterns between instances of attributes of $A(X)$ and $A(Y)$ in a derivation tree whose subtrees rooted in X and Y have been deleted. This set of graphs where X and Y are cousins, with X the "left cousin" and Y the "right cousin", will be called COUSIN-SET(X, Y).

The construction of an arbitrary graph from COUSIN-SET(X, Y) proceeds as follows. Consider an arbitrary pair $W, Z \in V$ (intuitively $X = W$ or X is a descendant of W and $Y = Z$ or Y is a descendant of Z). Select arbitrary graphs from the following sets: a graph brother (W, Z) from BROTHER-SET(W, Z) and graphs anc-desc(W, X) and anc-desc(Z, Y) from ANC-DESC-SET(W, X) and ANC-DESC-SET(Z, Y), respectively. The selected graphs are "pasted together" as pictured in Fig. 7.

With the resulting superposition of graphs a cousin graph can be associated as follows. For each pair of attributes $(X \cdot a, X \cdot b)$, $(X \cdot a, Y \cdot d)$, $(Y \cdot c, Y \cdot d)$ and $(Y \cdot c, X \cdot b)$, where $X \cdot a$ and $Y \cdot c$ are synthesized attributes, and $X \cdot b$ and $Y \cdot d$ are inherited attributes, an arc is included in the cousin graph if and only if there is an oriented path between the associated vertices in the superposition.

Parts a.4 and a.5: Consider a tree template tt with root Rtt and leaves Ltt_1, $Ltt_2, \ldots, Ltt_n$. For each instance T of tt we need its "instance graph" expressing the dependencies between attribute instances of $A(Rtt)$, $A(Ltt_1)$, $A(Ltt_2), \ldots, A(Ltt_n)$ in T.

First, from the dependency graph D_{tt} a "template graph" TG_{tt} can be derived as follows. The vertices of TG_{tt} are the attribute instances of the root and the leaves of tt. For each pair of attribute instances (a, tt, X) and (b, tt, Y), where (a, tt, X) is an input attribute instance and (b, tt, Y) is an output attribute instance, an arc $((a, tt, X), (b, tt, Y))$ is included in TG_{tt} if and only if there is an oriented path between the associated vertices in D_{tt}.

Next, we take into account the dependencies induced by the different instances of tt. This means that for each leaf $Ltt_i (1 \leqq i \leqq n)$ we need the set of graphs IS-SET(Ltt_i).

To construct an arbitrary instance graph of tt we choose an arbitrary is-graph from each set IS-SET(Ltt_i) $(1 \leqq i \leqq n)$. An instance graph can be constructed by taking the superposition of TG_{tt} and the selected is-graphs as pictured in Fig. 8.

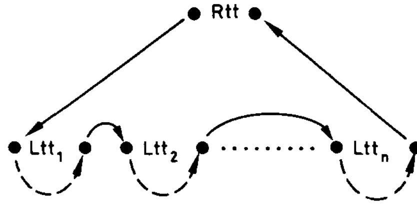

Fig. 8. Construction of an instance graph

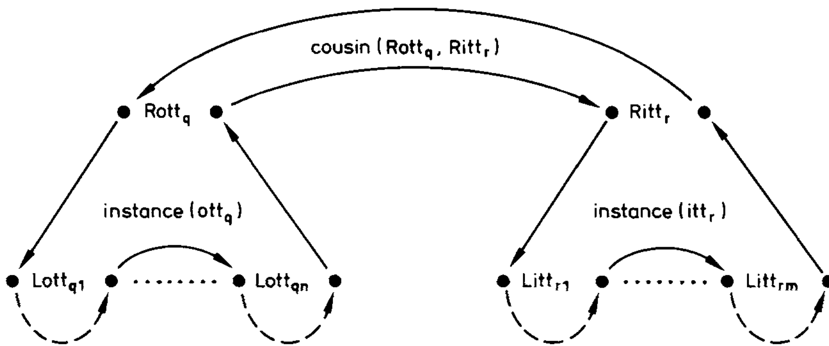

Fig. 9. Construction of a graph expressing dependencies between attribute instances of tree templates involved in a cousin relation

The set of all instance graphs of tt will be called INSTANCE-SET(tt) and can be constructed by considering all combinations of is-graphs from IS-SET(Ltt_i) $(1 \leqq i \leqq n)$.

Parts a.1–a.5.: Having available the necessary sets of cousin graphs and instance graphs we are now able to construct for every pair of tree templates ott_q and itt_r, for the case where their roots Rott_q and Ritt_r are cousins and where itt_r is found after ott_q, dependency graphs $\text{dg}(\text{ott}_q, \text{itt}_r)$ expressing dependency relations between attribute instances of ott_q and itt_r.

The construction of an arbitrary graph $\text{dg}(\text{ott}_q, \text{itt}_r)$ proceeds as follows. Select arbitrary graphs from the following sets: a graph $\text{cousin}(\text{Rott}_q, \text{Ritt}_r)$ from COUSIN-SET$(\text{Rott}_q, \text{Ritt}_r)$ and graphs $\text{instance}(\text{ott}_q)$ and $\text{instance}(\text{itt}_r)$ from INSTANCE-SET(ott_q) and INSTANCE-SET(itt_r), respectively. The selected graphs are pasted together as pictured in Fig. 9.

Up to now we ignored the labeling of the above-mentioned dependency graphs. Consider a derivation tree T including instances of output template ott_q and input template itt_r such that their roots are cousins and itt_r is found after ott_q. In Section 4 we explained how to assign a label L or $\bar{L}$ to every arc of D_T. Let $\text{dg}(\text{ott}_q, \text{itt}_r)$ be the dependency graph associated with T and constructed as pictured in Fig. 9. To each arc of $\text{dg}(\text{ott}_q, \text{itt}_r)$ a label is assigned in the following way. Each arc of $\text{dg}(\text{ott}_q, \text{itt}_r)$ is associated with one or more dependency paths in D_T. An arc in $\text{dg}(\text{ott}_q, \text{itt}_r)$ is labeled $\bar{L}$ if at least one

of its associated paths in D_T includes an arc labeled $\bar{L}$; otherwise the arc in dg(ott$_q$, itt$_r$) is labeled L. It will be shown in Sections 5.2–5.4 how to compute these labels, viz. by keeping track of these labels in all the graphs discussed up to now.

Case b: See Fig. 2b. Let T be a derivation tree including tree templates ott$_q$ and itt$_r$ such that Ritt$_r$ is a descendant of Rott$_q$ and itt$_r$ is found after ott$_q$. Let ott$_q$ have the leaves Lott$_{q1}$, ..., Lott$_{qn}$. The fact that ott$_q$ and itt$_r$ do not overlap means that ott$_q$ must have a leaf Lott$_{qk}$$(1 \leqq k \leqq n)$ such that Ritt$_r$ = Lott$_{qk}$ or Ritt$_r$ is a descendant of Lott$_{qk}$.

To construct the dependency graph expressing the dependency relations between attribute instances of EVAL(q) and COR(r) we need the following dependency relations in the following parts of T.

b.1) dependencies between attribute instances of ott$_q$ in the tree $T1$ obtained from T by deleting its subtree with root Lott$_{qk}$;

b.2) dependencies between attribute instances of A(Lott$_{qk}$) and A(Ritt$_r$) in the tree $T2$ obtained from T by isolating its subtree with root Lott$_{qk}$ and deleting its subtree with root Ritt$_r$.

b.3) dependencies between attribute instances of itt$_r$ in the tree $T3$ obtained from T by isolating its subtree with root Ritt$_r$.

As in Case a we do not restrict our computations to a specific derivation tree, but we are interested in any derivation tree including tree templates ott$_q$ and itt$_r$ such that Ritt$_r$ is a descendant of Rott$_q$ and itt$_r$ is found after ott$_q$.

In this section we make introductory remarks to each of the parts (b.1), (b.2) and (b.3). Algorithms to compute the necessary dependency relations are developed in the following sections.

Part b.1: Template graph TG$_{\text{ott}_q}$ describes the dependency relations of ott$_q$ isolated from its context. We also have to take into account the dependencies induced by the possible instances of ott$_q$ and their context, except for the subtree rooted at Lott$_{qk}$$(1 \leqq k \leqq n)$. Such a "context-except-for-one-leaf graph" (shortly a cefol graph) will be denoted by cefol(ott$_q$, k).

To construct an arbitrary graph cefol(ott$_q$, k) we choose an arbitrary *si*-graph from SI-SET(Rott$_q$) and an arbitrary *is*-graph from every IS-SET(Lott$_{qi}$) $(1 \leqq i \leqq n,\ i \neq k)$. Such a graph cefol(ott$_q$, k) can be constructed by pasting together these graphs as pictured in Fig. 10.

For a given k, the set of all graphs cefol(ott$_q$, k) will be called CEFOL-SET(ott$_q$, k).

Taking into account that R_{itt_r} can be a descendant of any Lott$_{qk}$ $(1 \leqq k \leqq n)$ we have to compute CEFOL-SET(ott$_q$, k) for each $k (1 \leqq k \leqq n)$.

Part b.2: For each pair (Lott$_{qk}$, Ritt$_r$) where Lott$_{qk}$$(1 \leqq k \leqq n)$ is a leaf of ott$_q$ and Ritt$_r$ is the root of itt$_r$ we need the set ANC-DESC-SET(Lott$_{qk}$, Ritt$_r$).

Part b.3: For itt$_r$ we need the set INSTANCE-SET(itt$_r$).

Parts b.1–b.3: Having available the necessary sets of dependency graphs we are now able to construct for each pair of tree templates ott$_q$ and itt$_r$, for the case where Ritt$_r$ is a descendant of Rott$_q$ and where itt$_r$ is found after

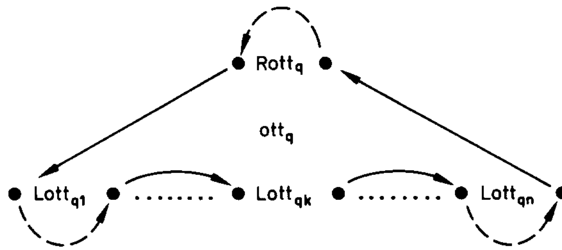

Fig. 10. Construction of a cefol graph

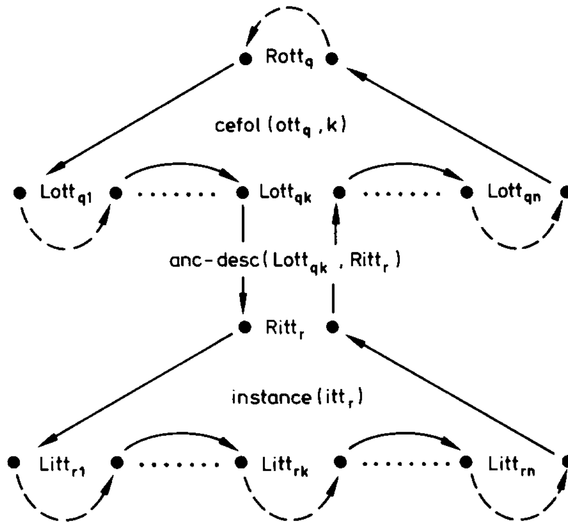

Fig. 11. Construction of a graph expressing dependencies between attribute instances of tree templates involved in an ancestor-descendant relation

ott_q, dependency graphs $\mathrm{dg}(\mathrm{ott}_q, \mathrm{itt}_r)$ expressing dependency relations between attribute instances of ott_q and itt_r.

The assemblage of an arbitrary graph $\mathrm{dg}(\mathrm{ott}_q, \mathrm{itt}_r)$ proceeds as follows. Take an arbitrary leaf Lott_{qk} of ott_q. Select arbitrary graphs from the following sets: a graph $\mathrm{cefol}(\mathrm{ott}_q, k)$ from $\text{CEFOL-SET}(\mathrm{ott}_q, k)$, a graph $\text{anc-desc}(\mathrm{Lott}_{qk}, \mathrm{Ritt}_r)$ from $\text{ANC-DESC-SET}(\mathrm{Lott}_{qk}, \mathrm{Ritt}_r)$ and a graph $\mathrm{instance}(\mathrm{itt}_r)$ from $\text{INSTANCE-SET}(\mathrm{itt}_r)$. The selected graphs are pasted together, as pictured in Fig. 11. In a similar way as for Case a a label has to be assigned to every arc.

Case c: See Fig. 2c. The set of all possible dependency graphs expressing the dependencies between attribute instances of $\mathrm{EVAL}(q)$ and $\mathrm{COR}(r)$, with ott_q and itt_r involved in a descendant-ancestor relation, can be found in the same way as for Case b.

Now we come back to the question whether for every derivation tree T and for each pair of tree transformation rules q: $(\mathrm{dir}_q, \mathrm{itt}_q, \mathrm{ott}_q, \mathrm{cond}_q, \mathrm{eval}_q)$

and r: (dir_r, itt_r, ott_r, cond_r, eval_r) such that itt_r is found after ott_q, r can be applied safely after q. Having available for each situation, as pictured in Fig. 2, the labeled graphs $\text{dg}(\text{ott}_q, \text{itt}_r)$ expressing the dependencies between attribute instances of ott_q and itt_r the question can be answered simply by checking the criteria of Theorem 4.1

In Section 5.2 we will discuss algorithms to compute IS-SET(X) and SI-SET(X) for each $X \in V$. In Section 5.3 we focus on the computation of BROTHER-SET(X, Y) for each pair X, $Y \in V$, FATHER-SON-SET(X, Y) and ANC-DESC-SET(X, Y) for each pair $X \in V_N$, $Y \in V$, and ANC-DESC-SET(X, X) for each $X \in V_T$. In Section 5.4 we demonstrate how to compute INSTANCE-SET(tt) and CEFOL-SET(tt, k) for every tree template tt and for every k. In Section 5.5 an algorithm will be presented to check for every pair q and r of conditional tree transformation rules whether r can be applied safely after q.

Observe that, although in general the number of derivation trees for a given grammar is infinite, the above-mentioned sets of graphs are finite, since the number of vertices is finite. However, the number of graphs in every set can be exponential in the size of the grammar. For this reason we also discuss a more "pessimistic" approach where every set of graphs is replaced by a single graph. Both methods are compared in Section 5.6.

For the direct expression of indirect dependencies we need the *transitive closure* of graphs with arcs labeled L or $\bar{L}$. Let D be a graph with vertices V and arcs A of which the arcs are labeled L or $\bar{L}$. The transitive closure D^* of D is the graph with vertices V, and arcs defined as follows. For a, $b \in V$ there is an arc (a, b) in D^* if and only if there is an oriented path from a to b in D. The arc (a, b) is labeled $\bar{L}$ if there is an oriented path from a to b in D of which at least one of the arcs is labeled $\bar{L}$, and L otherwise.

5.2 *Dependencies Between Attributes of a Single Tree Node*

In Section 4 for each production p a dependency graph DG_p has been defined. For the combination of the dependency information of a production and its context we introduce the following notation.

Let p be a production p: $X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn}$ and let $D_i (0 \leqq i \leqq n)$ be a directed graph with vertices $A(X_{pi})$ and arcs to which the label L or $\bar{L}$ is assigned. Then $\text{DG}_p[D_0, D_1, \ldots, D_n]$ is the directed graph obtained from DG_p by adding an arc from attribute occurrence (a, p, i) to (b, p, i) whenever there is an arc from attribute $X_{pi} \cdot a$ to attribute $X_{pi} \cdot b$ in $D_i (0 \leqq i \leqq n)$. The arc from (a, p, i) to (b, p, i) in $\text{DG}_p[D_0, D_1, \ldots, D_n]$ has the same label as its associated arc from $X_{pi} \cdot a$ to $X_{pi} \cdot b$ in $D_i (0 \leqq i \leqq n)$.

For the case where D_0 is an si-graph from SI-SET(X_{p0}) and $D_i (1 \leqq i \leqq n)$ an is-graph from IS-SET(X_{pi}) $(1 \leqq i \leqq n)$, $\text{DG}_p[D_0, D_1, \ldots, D_n]$ describes the (indirect) dependencies between attribute instances of an application of production p within a certain derivation tree. In the following we sometimes write $\text{DG}_p[\ldots]$ instead of $\text{DG}_p[D_0, D_1, \ldots, D_n]$ when the sequence of graphs to be used is clear from the context.

Now we consider the case of a "hole" in the context of a production. Given a production p: $X_{p0} \to X_{p1}\, X_{p2} \ldots X_{pn}$, let $D_i (0 \leq i \leq n)$ be a directed graph with vertices $A(X_{pi})$ and labeled arcs. Then for, $0 \leq k \leq n$, $DG_p - k[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n] = DG_p[D_0, D_1, \ldots, D_n]$, where D_k is the graph on $A(X_{pk})$ without edges. For $DG_p - k[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n]$ we also use the abbreviation $DG_p - k[\ldots]$.

For the direct expression of indirect dependencies we introduce $DG_p - k[\ldots]^*$, the transitive closure of $DG_p - k[\ldots]$. From $DG_p - k[\ldots]^*$ a directed graph $DG_p - k^*[\ldots]$ can be constructed as follows. The vertices of $DG_p - k^*[\ldots]$ are the attributes of $A(X_{pk})$. For each pair $X_{pk} \cdot a$, $X_{pk} \cdot b$ of attributes, an arc from $X_{pk} \cdot a$ to $X_{pk} \cdot b$ is included in $DG_p - k^*[\ldots]$ if and only if there is an arc from (a, p, k) to (b, p, k) in $DG_p - k[\ldots]^*$. Each arc in $DG_p - k^*[\ldots]$ has the same label as its corresponding arc in $DG_p - k[\ldots]^*$.

$DG_p - 0^*[\ldots]$ will be used to construct *is*-graphs from IS-SET(X_{p0}). $DG_p - k^*[\ldots]$ $(1 \leq k \leq n)$ will be used to obtain *si*-graphs from SI-SET(X_{pk}).

We discuss two methods to compute these labeled *is*- and *si*-graphs. The first one proceeds along the same lines as Knuth's correct algorithm [18] to examine the possible circularity of attribute grammars. The second one is related to Knuth's erroneous algorithm [17] and the variants of Kennedy and Warren [16] and Kastens [15]. The definition and computation of *is*- and *si*-graphs is also discussed in [20].

a. First Method. Consider a derivation tree T and its associated dependency graph D_T. D_T is the result of "pasting together" copies of DG_p's for productions applied in T. To each arc of D_T the same label L or $\bar{L}$ is assigned as to the associated arcs of the applied DG_p's.

In the following a subtree of T with root X will be denoted by T/X. With each subtree T/X a directed graph $IS_{T/X}$, representing the *i*-to-*s* behavior at X, can be associated as follows. The vertices of $IS_{T/X}$ are the attributes of $A(X)$. For each pair $X \cdot i$, $X \cdot s$ of inherited and synthesized attributes, respectively, an arc from $X \cdot i$ to $X \cdot s$ is included in $IS_{T/X}$ if and only if there is an oriented path between the associated vertices in $D_{T/X}$. To each arc of $IS_{T/X}$ a label is assigned in the following way. An arc of $IS_{T/X}$ is labeled $\bar{L}$ if at least one of its associated paths in $D_{T/X}$ includes an arc labeled $\bar{L}$; otherwise it is labeled L.

Now, for each $X \in V$ the set IS-SET(X) can be defined as follows:

Definition 5.1. For each $X \in V$, IS-SET$(X) = \{IS_{T/X} \mid T$ is a derivation tree$\}$. □

Observe that, for each $X \in V$, IS-SET(X) is finite since $A(X)$ is finite. An equivalent, more detailed, recursive definition, is:

Definition 5.2. For each $X \in V$,

$$\text{IS-SET}(X) = \{DG_p - 0^*[D_1, \ldots, D_n] \mid p \in P,\ X_{p0} = X,\ D_i \in \text{IS-SET}(X_{pi})\ (1 \leq i \leq n)\}. \quad \square$$

From this definition the following algorithm [18] can be derived.

Algorithm 5.1. Computation of the sets IS-SET(X).

Input: attribute grammar AG.

Output: set IS-SET(X) for all $X \in V$.

Algorithm:

Initial step:
for all $X \in V_N$ **do** IS-SET$(X) := \emptyset$ **od**;
for all $X \in V_T$
do IS-SET(X):=single graph with vertices $A(X)$ and no arcs **od**;
repeat {for all $X \in V_N$: add further graphs to set IS-SET(X)}
choose a production p: $X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn}$ for which none of the sets IS-SET(X_{pi}) $(1 \leqq i \leqq n)$ is empty;
for $1 \leqq i \leqq n$
do choose a graph D_i in IS-SET(X_{pi}) **od**;
if graph $\mathrm{DG}_p - 0^*[D_1, \ldots, D_n]$ is not in IS-SET(X_{p0})
then add this graph to IS-SET(X_{p0})
fi
until no further graphs can be added to any set IS-SET(X); □

Observe that, for all $X \in V$, IS-SET(X) is non-empty. This follows immediately from the fact that the underlying context-free grammar of AG is reduced.

Next we consider a derivation tree T whose subtree rooted in X has been deleted, except X itself. Such an (incomplete) derivation tree will be denoted by $T - T/X$.

With each tree $T - T/X$ a directed graph $\mathrm{SI}_{T-T/X}$, representing the s-to-i behavior at X, can be associated as follows. The vertices of $\mathrm{SI}_{T-T/X}$ are the attributes of $A(X)$. For each pair $X \cdot s$, $X \cdot i$ of synthesized and inherited attributes, respectively, an arc from $X \cdot s$ to $X \cdot i$ is included in $\mathrm{SI}_{T-T/X}$ if and only if there is an oriented path between the associated vertices in $D_{T-T/X}$. To each arc a label is assigned in the usual way, but note that in fact every arc is labeled $\bar{L}$.

Now, for each $X \in V$ the set SI-SET(X) can be defined as follows.

Definition 5.3. For each $X \in V$,

$$\text{SI-SET}(X) = \{\mathrm{SI}_{T-T/X} \mid T \text{ is a derivation tree}\}. \quad \square$$

An equivalent, more detailed, recursive definition is:

Definition 5.4. For each $X \in V$,

$$\begin{aligned}\text{SI-SET}(X) = \{&\mathrm{DG}_p - k^*[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n] \mid \\ &p \in P,\ 1 \leqq k \leqq n,\ X_{pk} = X,\ D_0 \in \text{SI-SET}(X_{p0}), \\ &D_i \in \text{IS-SET}(X_{pi})\ (1 \leqq i \leqq n,\ i \neq k)\}. \quad \square\end{aligned}$$

This definition leads to the following algorithm.

Algorithm 5.2. Computation of the sets SI-SET(X).

Input: attribute grammar AG; set IS-SET(X) for all $X \in V$.

Output: set SI-SET(X) for all $X \in V$.

Algorithm:

Initial step:
SI-SET(S):=single graph with vertices $A(S)$ and no arcs;
for all $X \in V$ such that $X \neq S$ **do** SI-SET(X):=$\emptyset$ **od**;
repeat {for all $X \in V$: add further graphs to set SI-SET(X)}
choose a production p: $X_{p0} \to X_{p1} X_{p2} \ldots X_{pn}$ for which the set SI-SET(X_{p0}) is not empty;
choose a graph D_0 in SI-SET(X_{p0});
choose an integer $k(1 \leqq k \leqq n)$;
for $1 \leqq i \leqq n$, $i \neq k$
do choose a graph D_i in IS-SET(X_{pi}) **od**;
if graph $\mathrm{DG}_p - k^*[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n]$ is not in SI-SET(X_{pk})
then add this graph to SI-SET(X_{pk})
fi
until no further graphs can be added to any set SI-SET(X); □

Algorithm 5.1 to compute the sets IS-SET(X) for $X \in V$ is the essential portion of Knuth's algorithm to examine the possible circularity of attribute grammars [18]. Algorithm 5.2 to compute the sets SI-SET(X) is just a simple variation. Jazayeri, Ogden and Rounds [12, 13] showed that the time complexity of the circularity test is inherently exponential. However, the only exponential factor in the complexity of the algorithm is the number of *is*-graphs in the set constructed for each nonterminal symbol. Räihä and Saarinen [22] found that for practical grammars this number is very small (at most 4 for unlabeled graphs) and discussed some techniques to improve the implementation of the algorithm to compute the sets IS-SET(X). Their experiments showed that for practical grammars the computation of the dependencies is feasible.

b. Second Method. For a safety-test algorithm that is polynomial the sets IS-SET(X) and SI-SET(X) have to be replaced by single dependency graphs IS_X and SI_X, respectively. For each $X \in V_N$ the graph IS_X will contain (at least) all arcs of all different *is*-graphs of all different subtrees T/X of all different derivation trees T (in the sense that $\bar{L}$ overrides L). For each $X \in V$ the graph SI_X will represent a mixture of the different *si*-graphs of all different trees $T - T/X$.

For each $X \in V$ the graph IS_X can be defined recursively as follows.

Definition 5.5 For each $X \in V$,

$$\mathrm{IS}_X = \cup \{\mathrm{DG}_p - 0^*[\mathrm{IS}_{X_{p1}}, \ldots, \mathrm{IS}_{X_{pn}}] \mid p \in P, X_{p0} = X\}. \quad \square$$

In this definition the union operator "$\cup$" has the following meaning. A certain arc is included in IS_X if an only if there is an associated arc in at least one of the graphs $\mathrm{DG}_p - 0^*[\ldots]$, where $X_{p0} = X$. An arc in IS_X is labeled $\bar{L}$ if at least one of the associated arcs is labeled $\bar{L}$; otherwise it is labeled L.

From Definition 5.5 the following algorithm [16, 17] can be derived.

Algorithm 5.3. Computation of dependency graphs IS_X.

Input: attribute grammar AG.

Output: graph IS_X for all $X \in V$.

Algorithm:
  Initial step:
    for all $X \in V$
    do IS_X:=graph with vertices $A(X)$ and no arcs **od**;
  repeat {for all $X \in V_N$: add further arcs to graph IS_X
      and change the label of arcs already in IS_X}
    choose a production p: $X_{p0} \to X_{p1}\, X_{p2} \ldots X_{pn}$;
    if $DG_p - 0^*[IS_{X_{p1}}, \ldots, IS_{X_{pn}}]$ includes an arc
      $(X_{p0} \cdot a, X_{p0} \cdot b)$ not in $IS_{X_{p0}}$
    then add this arc to $IS_{X_{p0}}$
    fi;
    if $DG_p - 0^*[IS_{X_{p1}}, \ldots, IS_{X_{pn}}]$ includes an arc
      $(X_{p0} \cdot a, X_{p0} \cdot b)$ labeled $\bar{L}$
      while the corresponding arc in $IS_{X_{p0}}$ is labeled L
    then change the label of arc $(X_{p0} \cdot a, X_{p0} \cdot b)$ in $IS_{X_{p0}}$
      from L into $\bar{L}$
    fi
  until no further arcs can be added to any graph IS_X
    and no more labels can be changed in any graph IS_X; □

In [16] it has been shown that the (unlabeled) graphs IS_X, by mixing *i*-to-*s* dependencies of different subtrees with the same root X, not only may exhibit combinations of dependencies that never will occur together in a single graph of the set IS-SET(X), but also may suggest spurious dependencies (i.e., dependencies that never will occur in any subtree with root X). Also the labeling of IS_X may be pessimistic, in the following sense. Let *is*-graph be a graph in IS-SET(X). If arc(a, b) of *is*-graph has label L, then arc(a, b) of IS_X has label L or $\bar{L}$. If arc(a, b) of *is*-graph has label $\bar{L}$, then arc(a, b) of IS_X has label $\bar{L}$.

Similarly, it can be shown that the graphs SI_X, by mixing *s*-to-*i* dependencies associated with different trees $T - T/X$, may exhibit dependencies that never will occur together in a single graph of the set SI-SET(X) and even may suggest dependencies that never will occur for any tree $T - T/X$.

Starting from the following definition the development of an algorithm to compute the sets SI_X for all $X \in V$ is straightforward and is left to the reader.

Definition 5.6. For each $X \in V$,

$$SI_X = \cup \{DG_p - k^*[SI_{X_{p0}}, IS_{X_{p1}}, \ldots, IS_{X_{p(k-1)}}, IS_{X_{p(k+1)}}, \ldots, IS_{X_{pn}}] \mid p \in P,\ 1 \leq k \leq n,\ X_{pk} = X\}. \quad \square$$

Since the sets IS_X and SI_X present a sort of worst-case pictures of IS and SI dependencies, respectively, they may mistakenly suggest that the correct recomputation of necessary attributes is not possible during a single pass between two transformations. But, a positive conclusion is always correct. Investigation of practical examples has to show whether the use of graphs from the sets IS-SET(X) and SI-SET(X) really leads to better results and hence is preferable to the use of graphs IS_X and SI_X.

To aid in performance analyis a more detailed representation of Algorithm 5.3 is included. In Algorithm 5.3a, instead of $DG_p-0^*[\ldots]$ we construct the transitive closure $DG_p-0[\ldots]^*$ of $DG_p-0[\ldots]$.

Algorithm 5.3a. Computation of dependency graphs IS_X.

Input: attribute grammar AG.

Output: graph IS_X for all $X \in V$.

Algorithm:

```
for all X∈V
do construct graph IS_X with vertices A(X) and no arcs od;
for all p∈P
do initialize graph DG_p−0[...]* = DG_p−0[...];
   for each arc(X_p0·a, X_p0·b) in DG_p−0[...]*
       not in IS_{X_p0}
       or in IS_{X_p0} but labeled differently
   do add arc(X_p0·a, X_p0·b) to IS_{X_p0}
      or adjust the label of arc(X_p0·a, X_p0·b) if necessary
   od
od;
{initially, for each X∈V_N each arc in IS_X is unmarked}
while there is an X∈V_N with an unmarked arc(X·a, X·b) in IS_X
do mark(X·a, X·b) in IS_X;
   for each occurrence of X = X_pk(1≦k≦n)
       in the right-hand side of any rule p
   do add arc(X_pk·a, X_pk·b) to DG_p−0[...]*;
      update the transitive closure DG_p−0[...]*;
      for each arc(X_p0·a, X_p0·b) in DG_p−0[...]*
          not in IS_{X_p0}
          or in IS_{X_p0} but labeled differently
      do add arc(X_p0·a, X_p0·b) to IS_{X_p0}
         or (adjust the label of arc(X_p0·a, X_p0·b)
             and unmark (X_p0·a, X_p0·b)) if necessary
      od
   od
od; □
```

To express the time complexity of Algorithm 5.3a we introduce the following parameters:

$|V|$ the number of grammar symbols;
$|P|$ the number of productions;
$|R|$ the maximum number of grammar symbols in a single production;
$|X|$ the maximum number of attributes of a single grammar symbol.

The inclusion "add arc(a, b) to graph D", the adaptations "adjust the label of arc(a, b)" and "mark or unmark arc(a, b)" and the test "arc(a, b) in graph D" are considered as primitive operations. If the costs of these operations is $O(1)$, then the time complexity of Algorithm 5.3a is $O(|V||P||R|^4|X|^5)$, i.e., the time complexity is polynomial in the size of the grammar. For a similar algorithm to compute the sets SI_X for each $X \in V_N$ the same complexity is found.

5.3 Dependencies Between Attributes of Two Tree Nodes

In Sect. 5.2 we introduced the concept of a "hole in the context" of a production. For the computation of the data flow in a chain of productions, we also need the idea of a "context with two holes", since a production usually has two neighbours in a chain.

Given a production p: $X_{p0} \to X_{p1} X_{p2} \dots X_{pn}$ and directed graphs D_i ($0 \leqq i \leqq n$) with vertices $A(X_{pi})$ and arcs to which the label L or $\bar{L}$ is assigned. For $0 \leqq j \leqq k \leqq n$,

$$\mathrm{DG}_p - (j, k)\,[D_0, D_1, \dots, D_{j-1}, D_{j+1}, \dots, D_{k-1}, D_{k+1}, \dots, D_n] = \mathrm{DG}_p[D_0, D_1, \dots, D_n],$$

where D_j and D_k are the graphs on $A(X_{pj})$ and $A(X_{pk})$, respectively, both without edges. For

$$\mathrm{DG}_p - (j, k)\,[D_0, D_1, \dots, D_{j-1}, D_{j+1}, \dots, D_{k-1}, D_{k+1}, \dots, D_n]$$

we also use the abbreviation $\mathrm{DG}_p - (j, k)\,[\dots]$.

For the direct expression of indirect dependencies we use $\mathrm{DG}_p - (j, k)\,[\dots]^*$, the transitive closure of $\mathrm{DG}_p - (j, k)\,[\dots]$.

From $\mathrm{DG}_p - (j, k)\,[\dots]^*$ a directed graph $\mathrm{DG}_p - (j, k)^*[\dots]$ can be constructed as follows. The vertices of $\mathrm{DG}_p - (j, k)^*[\dots]$ are the attributes of $A(X_{pj})$ and $A(X_{pk})$. A certain arc is included in $\mathrm{DG}_p - (j-k)^*[\dots]$ if and only if there is an associated arc in $\mathrm{DG}_p - (j, k)\,[\dots]^*$. Each arc in $\mathrm{DG}_p - (j, k)^*[\dots]$ has the same label as its corresponding arc in $\mathrm{DG}_p - (j, k)\,[\dots]^*$. Observe that in $\mathrm{DG}_p - (j, k)^*[\dots]$ the following kinds of arcs have to be distinguished: arcs $(X_{pj} \cdot a, X_{pj} \cdot b)$, $(X_{pk} \cdot c, X_{pk} \cdot a)$, $(X_{pj} \cdot a, X_{pk} \cdot d)$ and $(X_{pk} \cdot c, X_{pj} \cdot b)$ associated with arcs $((a, p, j), (b, p, j))$, $((c, p, k), (d, p, k))$, $((a, p, j), (d, p, k))$ and $((c, p, k), (b, p, j))$, respectively, where (a, p, j) and (c, p, k) are used attribute occurrences and (b, p, j) and (d, p, k) are defined attribute occurrences of production p.

Remark. In a graph $\mathrm{DG}_p - (j, k)^*[\dots]$, where $X = X_{pj} = X_{pk}$, two instances of each attribute $a \in A(X)$ are involved. Subscripts are used to distinguish the two instances of each attribute $a \in A(X)$.

$DG_p-(j,k)^*[\ldots]$ $(1 \leqq j < k \leqq n)$ will be used to express brother graphs from the set BROTHER-SET(X_{pj}, X_{pk}). $DG_p-(0,k)^*[\ldots]$ $(1 \leqq k \leqq n)$ will be used to express father-son graphs from the set FATHER-SON-SET(X_{p0}, X_{pk}).

As in Sect. 5.2 we discuss two methods to compute brother, father-son, ancestor-descendant and cousin graphs. The first method uses sets of different dependency graphs and is related to Knuth's correct algorithm [18]. The second method mixes the dependencies of different graphs into a single graph and is related to Knuth's erroneous algorithm [17].

a. First Method. We first discuss the computation of brother-graphs, starting from the following (detailed) definition.

Definition 5.7. For each pair $X, Y \in V$,

$$\text{BROTHER-SET}(X, Y) = \{DG_p-(j,k)^*[D_0, D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n] \mid p \in P,\ 1 \leqq j < k \leqq n,\ X_{pj}=X,\ X_{pk}=Y,\ D_0 \in \text{SI-SET}(X_{p0}),\ D_i \in \text{IS-SET}(X_{pi})\ (1 \leqq i \leqq n,\ i \neq j, k)\}. \quad \square$$

This definition immediately leads to the following algorithm to compute the sets BROTHER-SET(X, Y) for all $X, Y \in V$.

Algorithm 5.4 Computation of sets BROTHER-SET(X, Y).

Input: attribute grammar AG; sets IS-SET(X) and SI-SET(X) for all $X \in V$.

Output: set BROTHER-SET(X, Y) for all $X, Y \in V$.

Algorithm:

Initial step:

```
        for all X, Y∈V
        do BROTHER-SET(X, Y):=∅ od;
    {for all X, Y∈V: add further graphs to set BROTHER-SET(X, Y)}
    for all p∈P
    do for each pair j, k(1≦j<k≦n)
        do for each combination
            [D0, D1, ..., Dj-1, Dj+1, ..., Dk-1, Dk+1, ..., Dn]
            where D0∈SI-SET(Xp0) and
                  Di∈IS-SET(Xpi) (1≦i≦n, i≠j, k)
            do if graph DGp-(j, k)* [D0, D1, ..., Dj-1, Dj+1, ..., Dk-1,
                  Dk+1, ..., Dn]
                  is not in BROTHER-SET(Xpj, Xpk)
               then add this graph to BROTHER-SET(Xpj, Xpk)
               fi
            od
        od
    od; □
```

Next, we focus on the computation of father-son graphs.

Definition 5.8. For each pair $X \in V_N$, $Y \in V$,

$$\text{FATHER-SON-SET}(X, Y) = \{DG_p - (0, k)^*[D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n] \mid p \in P, X_{p0} = X, 1 \leqq k \leqq n, X_{pk} = Y, D_i \in \text{IS-SET}(X_{pi}) \ (1 \leqq i \leqq n, i \neq k)\}. \quad \square$$

Starting from this definition the development of an algorithm to compute the sets FATHER-SON-SET(X, Y) for all $X \in V_N$, $Y \in V$, is straightforward and is left to the reader.

In Sect. 5.1 it has been explained how to compute the ancestor-descendant graphs from sequences of father-son graphs. The following algorithm may be used to compute the sets ANC-DESC-SET(X, Y) for all $X \in V_N$, $Y \in V$ and the sets ANC-DESC-SET(X, X) for all $X \in V_T$.

Algorithm 5.5 Computation of sets ANC-DESC-SET(X, Y).

Input: attribute grammar AG;
set FATHER-SON-SET(X, Y) for all $X \in V_N$, $Y \in V$.

Output: set ANC-DESC-SET(X, Y) for all $X \in V_N$, $Y \in V$ and
set ANC-DESC-SET(X, X) for all $X \in V_T$.

Algorithm:

Initial step:
  for all $X \in V_N$, $Y \in V$
  do ANC-DESC-SET(X, Y):=FATHER-SON-SET(X, Y) **od**;
  for all $X \in V_T$
  do ANC-DESC-SET(X, X):=single graph ident$(X_{\text{anc}}, X_{\text{desc}})$ **od**;
  for all $X \in V_N$
  do if ident$(X_{\text{anc}}, X_{\text{desc}})$ is not in ANC-DESC-SET(X, X)
    then add this graph to ANC-DESC-SET(X, X)
    fi
  od;
repeat {for all $X \in V_N$, $Y \in V$: add further graphs to set ANC-DESC-SET(X, Y)}
  choose $X, Z \in V_N$ and $Y \in V$ such that ANC-DESC-SET(X, Z) and ANC-DESC-SET(Z, Y) are not empty;
  choose a graph anc-desc(X, Z) from ANC-DESC-SET(X, Z);
  choose a graph anc-desc(Z, Y) from ANC-DESC-SET(Z, Y);
  construct the graph superposition (X, Z, Y) by "pasting together along Z" the graphs anc-desc(X, Z) and anc-desc(Z, Y);
  construct the graph anc-desc(X, Y) with vertices $A(X)$ and $A(Y)$ and labeled arcs $(X \cdot a, X \cdot b)$, $(X \cdot a, Y \cdot d)$, $(Y \cdot c, Y \cdot d)$ and $(Y \cdot c, X \cdot b)$ if and only if there are associated arcs in superposition $(X, Z, Y)^*$;
  if anc-desc (X, Y) is not in ANC-DESC-SET(X, Y)
  then add this graph to ANC-DESC-SET(X, Y)
  fi
until no further graphs can be added to any set ANC-DESC-SET(X, Y); $\square$

Observe that, for all $X \in V$, the identity graph ident(X_{anc}, X_{desc}) includes two kinds of arcs: $(X_{anc} \cdot i, X_{desc} \cdot i)$ and $(X_{desc} \cdot s, X_{anc} \cdot s)$ for i and s inherited and synthesized attributes, respectively. These arcs are labeled L.

Having available the necessary sets of brother graphs and ancestor-descendant graphs the following algorithm may be used to compute the set COUSIN-SET(X, Y) for all $X, Y \in V$.

Algorithm 5.6. Computation of sets COUSIN-SET(X, Y).

Input: attribute grammar AG;
set BROTHER-SET(X, Y) for all $X, Y \in V$;
set ANC-DESC-SET(X, Y) for all $X \in V_N$, $Y \in V$;
set ANC-DESC-SET(X, X) for all $X \in V_T$.

Output: set COUSIN-SET(X, Y) for all $X, Y \in V$.

Algorithm:

Initial step:

```
        for all X, Y∈V
        do COUSIN-SET(X, Y):=BROTHER-SET(X, Y) od;
    {for all X, Y∈V: add further graphs to set COUSIN-SET(X, Y)}
    for all X, Y∈V
    do for each pair W, Z∈V
           such that BROTHER-SET(W, Z) is non-empty, and both ANC-
           DESC-SET(W, X) and ANC-DESC-SET(Z, Y) exist and are non-
           empty
        do for all graphs brother(W, Z) from BROTHER-SET(W, Z)
           do for all graphs anc-desc(W, X) from ANC-DESC-SET(W, X)
              do for all graphs anc-desc(Z, Y) from ANC-DESC-SET(Z, Y)
                 do construct the graph superposition (X, W, Z, Y) by "pasting
                    together along W" the graphs brother(W, Z) and anc-
                    desc(W, X) and "along Z" the graphs brother(W, Z) and
                    anc-desc(Z, Y);
                    construct the graph cousin(X, Y) with vertices A(X) and
                    A(Y) and labeled arcs (X·a, X·b), (X·a, Y·d), (Y·c, Y·d)
                    and (Y·c, X·b) if and only if there are associated arcs in
                    superposition (X, W, Z, Y)*;
                    if cousin(X, Y) is not in COUSIN-SET(X, Y)
                    then add this graph to COUSIN-SET(X, Y)
                    fi
                 od
              od
           od
        od
    od; □
```

From the fact that the number of graphs in the sets IS-SET(X) and SI-SET(X) is exponential follows that the number of graphs in the sets BROTHER-SET(X, Y), FATHER-SON-SET(X, Y), ANC-DESC-SET(X, Y) and COUSIN-SET(X, Y) is also exponential.

Restricting the number of graphs in the sets IS-SET(X) and SI-SET(X) to one reduces the number of graphs in the sets BROTHER-SET(X, Y) and FATHER-SON-SET (X, Y) to an amount polynomial in the size of the grammar, but the number of graphs in the sets ANC-DESC-SET(X, Y) may still be exponential. From this we conclude that also the number of graphs in the sets COUSIN-SET(X, Y) may still be exponential.

b. Second Method. To obtain a safety-test algorithm that is polynomial we already replaced the sets of graphs IS-SET(X) and SI-SET(X) by the single graphs IS_X and SI_X, respectively. Moreover, instead of the sets of graphs BROTHER-SET(X, Y), FATHER-SON-SET(X, Y), ANC-DESC-SET(X, Y) and COUSIN-SET(X, Y) we will use single graphs $BROTHER_{X,Y}$, $FATHER\text{-}SON_{X,Y}$, $ANC\text{-}DESC_{X,Y}$ and $COUSIN_{X,Y}$.

For each pair $X, Y \in V$ the graph $BROTHER_{X,Y}$ represents a mixture of the different brother-graphs from BROTHER-SET(X, Y).

Definition 5.9. For each pair $X, Y \in V$,

$$BROTHER_{X,Y} = \cup \{DG_p - (j,k)^*[SI_{X_{p0}}, IS_{X_{p1}}, \ldots, IS_{X_{p(j-1)}}, IS_{X_{p(j+1)}} \ldots, IS_{X_{p(k-1)}}, IS_{X_{p(k+1)}}, \ldots, IS_{X_{pn}}] \mid p \in P, 1 \leq j < k \leq n, X_{pj} = X, X_{pk} = Y\}. \quad \square$$

In this definition the union operator "$\cup$" has the usual meaning of joining arcs of different graphs, in the sense that $\bar{L}$ overrides L (cf. Def. 5.5).

This definition immediately leads to the following algorithm:

Algorithm 5.7. Computation of dependency graphs $BROTHER_{X,Y}$.

Input: attribute grammar AG; graphs IS_X and SI_X for all $X \in V$.

Output: graph $BROTHER_{X,Y}$, for all $X, Y \in V$.

Algorithm:

Initial step:
    for all $X, Y \in V$
    do $BROTHER_{X,Y}$:=graph with vertices $A(X)$ and $A(Y)$ and no arcs **od**;
{for all $X, Y \in V$: add further arcs to graph $BROTHER_{X,Y}$
 and change the label of arcs already in $BROTHER_{X,Y}$}
for all $p \in P$
do for each pair $j, k (1 \leq j < k \leq n)$
    do if $DG_p - (j,k)^*[SI_{X_{p0}}, IS_{X_{p1}}, \ldots, IS_{X_{p(j-1)}}, IS_{X_{p(j+1)}}, \ldots, IS_{X_{p(k-1)}}, IS_{X_{p(k+1)}}, \ldots, IS_{X_{pn}}]$
        includes arcs $(X_{pj} \cdot a, X_{pj} \cdot b)$, $(X_{pj} \cdot a, X_{pk} \cdot d)$,
        $(X_{pk} \cdot c, X_{pk} \cdot d)$ and $(X_{pk} \cdot c, X_{pj} \cdot b)$
        not in $BROTHER_{X_{pj},X_{pk}}$
        or in $BROTHER_{X_{pj},X_{pk}}$ but labeled differently
      then add these arcs to $BROTHER_{X_{pj},X_{pk}}$
          or adjust their label if necessary
      fi
    od
od; $\square$

In Sect. 5.2 it was shown that the graphs IS_X and SI_X may mix-up dependencies associated with different environments and may even picture spurious dependencies. Hence, also the graph $BROTHER_{X,Y}$ may exhibit dependencies that never will occur together in any single graph of the set BROTHER-SET(X, Y) and may even suggest dependencies (and also labels $\bar{L}$ instead of L) that never will occur for any environment of the pair X, Y. Similar remarks can be made for the graphs $\text{FATHER-SON}_{X,Y}$, $\text{ANC-DESC}_{X,Y}$ and $\text{COUSIN}_{X,Y}$.

To express the time complexity of Algorithm 5.7, we use the same parameters and cost values for primitive operations as defined at the end of Sect. 5.2. Assuming the availability of the graphs IS_X and SI_X for all $X \in V$, Algorithm 5.7 takes time $O(|P||R|^5|X|^3)$, i.e., the time complexity is polynomial in the size of the grammar.

For each pair $X \in V_N$, $Y \in V$ the graph $\text{FATHER-SON}_{X,Y}$ representing a mixture of the different father-son graphs from the set FATHER-SON-SET(X, Y), can be defined as follows.

Definition 5.10. For each pair $X \in V_N$, $Y \in V$,

$$\text{FATHER-SON}_{X,Y} = \cup \{DG_p - (0, k)^* [IS_{X_{p1}}, \ldots, IS_{X_{p(k-1)}}, IS_{X_{p(k+1)}}, \ldots, IS_{X_{pn}}] \mid p \in P, X_{p0} = X, 1 \leqq k \leqq n, X_{pk} = Y\}. \quad \square$$

Starting from this definition the development of an algorithm to compute the graph $\text{FATHER-SON}_{X,Y}$, for all $X \in V_N$, $Y \in V$, is straightforward and is left to the reader. Observe that the time complexity of such an algorithm is $O(|P||R|^4|X|^3)$.

From the father-son graphs we construct for all $X \in V_N$, $Y \in V$ the graph $\text{ANC-DESC}_{X,Y}$ and for all $X \in V_T$ the graph $\text{ANC-DESC}_{X,X}$.

Algorithm 5.8. Computation of dependency graphs $\text{ANC-DESC}_{X,Y}$.

Input: attribute grammar AG; graph $\text{FATHER-SON}_{X,Y}$ for all $X \in V_N$, $Y \in V$.

Output: graph $\text{ANC-DESC}_{X,Y}$ for all $X \in V_N$, $Y \in V$ and
graph $\text{ANC-DESC}_{X,X}$ for all $X \in V_T$.

Algorithm:

Initial step:
for all $X \in V_N$, $Y \in V$
do $\text{ANC-DESC}_{X,Y} := \text{FATHER-SON}_{X,Y}$ **od**;
for all $X \in V_T$
do $\text{ANC-DESC}_{X,X} := \text{ident}(X_{\text{anc}}, X_{\text{desc}})$ **od**;
for all $X \in V_N$
do if $\text{ident}(X_{\text{anc}}, X_{\text{desc}})$ includes arcs not in $\text{ANC-DESC}_{X,X}$
then add these arcs to $\text{ANC-DESC}_{X,X}$
fi
od;
{for all $X \in V_N$, $Y \in V$: add further arcs to graph $\text{ANC-DESC}_{X,Y}$ and change the label of arcs already in $\text{ANC-DESC}_{X,Y}$}
for all X, $Y \in V_N$, $Z \in V$

do construct the graph superposition (X, Y, Z) by "pasting together along
 Y" the graphs $\text{ANC-DESC}_{X,Y}$ and $\text{ANC-DESC}_{Y,Z}$
 and construct its transitive closure superposition $(X, Y, Z)^*$
od;
{initially for all $X \in V_N$, $Y \in V$ every arc in $\text{ANC-DESC}_{X,Y}$ is unmarked}
while there are $X \in V_N$, $Y \in V$
  such that graph $\text{ANC-DESC}_{X,Y}$ has unmarked arcs
do mark the arcs of $\text{ANC-DESC}_{X,Y}$;
 for each $Z \in V$
 do if $Y \in V_N$
  then update superposition $(X, Y, Z)^*$;
   for each arc $(X \cdot a, X \cdot b)$, $(X \cdot a, Z \cdot d)$, $(Z \cdot c, Z \cdot d)$ and
    $(Z \cdot c, X \cdot b)$ in superposition $(X, Y, Z)^*$
    not in $\text{ANC-DESC}_{X,Z}$
    or in $\text{ANC-DESC}_{X,Z}$ but labeled differently
   do add these (unmarked) arcs to $\text{ANC-DESC}_{X,Z}$
    or adjust their label and unmark them, if necessary
   od
  fi;
  if $Z \in V_N$
  then update superposition $(Z, X, Y)^*$;
   {same as above: add arcs to and unmark arcs
     and adjust labels of arcs in $\text{ANC-DESC}_{Z,Y}$}
  fi
 od
od; □

Using the same parameters and the same primitive operations with associated cost values as at the end of Sect. 5.2 the time complexity of Algorithm 5.8 is $O(|V|^3|X|^4)$.

Finally, we present Algorithm 5.9 to construct the graph $\text{COUSIN}_{X,Y}$ for all $X, Y \in V$.

Algorithm 5.9. Computation of dependency graphs $\text{COUSIN}_{X,Y}$.

Input: attribute grammar AG;
 graph $\text{BROTHER}_{X,Y}$ for all $X, Y \in V$;
 graph $\text{ANC-DESC}_{X,Y}$ for all $X \in V_N$, $Y \in V$;
 graph $\text{ANC-DESC}_{X,X}$ for all $X \in V_T$.
Output: graph $\text{COUSIN}_{X,Y}$ for all $X, Y \in V$.

Algorithm:
 Initial step:
  for all $X, Y \in V$
  do $\text{COUSIN}_{X,Y} := \text{BROTHER}_{X,Y}$ **od**;
 {for all $X, Y \in V$: add further arcs to graph $\text{COUSIN}_{X,Y}$
  and change the label of arcs already in $\text{COUSIN}_{X,Y}$}

for all X, $Y \in V$
do **for** each pair W, $Z \in V$
such that both $\text{ANC-DESC}_{W,X}$ and $\text{ANC-DESC}_{Z,Y}$ exist
do construct the graph superposition (X, W, Z, Y) by "pasting together along W" the graphs $\text{BROTHER}_{W,Z}$ and $\text{ANC-DESC}_{W,X}$ and "along Z" the graphs $\text{BROTHER}_{W,Z}$ and $\text{ANC-DESC}_{Z,Y}$ and construct its transitive closure superposition $(X, W, Z, Y)^*$;
for each arc $(X \cdot a, X \cdot b)$, $(X \cdot a, Y \cdot d)$, $(Y \cdot c, Y \cdot d)$ and $(Y \cdot c, X \cdot b)$ in superposition $(X, W, Z, Y)^*$
not in $\text{COUSIN}_{X,Y}$
or in $\text{COUSIN}_{X,Y}$ but labeled differently
do add these arcs to $\text{COUSIN}_{X,Y}$
or adjust their label if necessary
od
od
od; □

Using the same parameters and cost values for primitive operations as at the end of Sect. 5.2, the time complexity of Algorithm 5.9 is $O(|V|^4 |X|^3)$.

5.4 Context Dependencies for Tree Templates

In Sect. 5.1 the notion of a template graph TG_{tt} of a tree template tt has been defined. Now we add labels to the arcs in the usual manner. For the combination of the dependency information of a tree template and its context we introduce a notation similar to the one used for productions.

Given a tree template tt with root Rtt and leaves $Ltt_1, Ltt_2, \ldots, Ltt_n$. Let D_0 be a graph from SI-SET(Rtt) and let $D_i (1 \leqq i \leqq n)$ be a graph from IS-SET(Ltt_i). Then $\text{TG}_{tt}[D_0, D_1, \ldots, D_n]$ is the directed graph obtained from TG_{tt} by adding an arc from attribute instance (a, tt, Rtt) to (b, tt, Rtt) whenever there is an arc from attribute $Rtt \cdot a$ to $Rtt \cdot b$ in D_0 and by adding an arc from attribute instance (a, tt, Ltt_i) to (b, tt, Ltt_i) whenever there is an arc from attribute $Ltt_i \cdot a$ to $Ltt_i \cdot b$ in $D_i (1 \leqq i \leqq n)$. The arcs added to TG_{tt} have the same label as their associated arcs in $D_i (0 \leqq i \leqq n)$.

Consider Fig. 2. Observe that both output template ott_q and input template itt_r have one neighbour in the chain of connecting productions. So, also for a tree template, we need the concept of a "hole" in its context.

$\text{TG}_{tt} - 0[D_1, \ldots, D_n] = \text{TG}_{tt}[D_0, D_1, \ldots, D_n]$, where D_0 is the graph on $A(Rtt)$ without edges. For $1 \leqq k \leqq n$, $\text{TG}_{tt} - k[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n] = \text{TG}_{tt}[D_0, D_1, \ldots, D_n]$, where D_k is the graph on $A(Ltt_k)$ without edges. $\text{TG}_{tt} - 0[D_1, D_2, \ldots, D_n]$ will be used to express instance graphs, and $\text{TG}_{tt} - k[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n]$ $(1 \leqq k \leqq n)$ to express cefol graphs.

As usual we discuss two methods to compute the instance graphs and cefol graphs for each tree template. The first approach results in sets of different graphs. The second method mixes the dependencies of different environments into a single graph.

a. First Method

Definition 5.11. For each tree template tt with leaves $Ltt_1, Ltt_2, \ldots, Ltt_n$, INSTANCE-SET$(tt) = \{TG_{tt} - 0[D_1, D_2, \ldots, D_n] \mid D_i \in$ IS-SET(Ltt_i) $(1 \leqq i \leqq n)\}$. □

Definition 5.12. For each tree template tt with root Rtt and leaves Ltt_1, $Ltt_2, \ldots, Ltt_n$ and integer $k (1 \leqq k \leqq n)$,

$$\text{CEFOL-SET}(tt, k) = \{TG_{tt} - k[D_0, D_1, \ldots, D_{k-1}, D_{k+1}, \ldots, D_n] \mid D_0 \in \text{SI-SET}(Rtt), D_i \in \text{IS-SET}(Ltt_i)\ (1 \leqq i \leqq n, i \neq k)\}. \quad \square$$

Algorithms to compute the above-mentioned sets follow immediately from Definitions 5.11 and 5.12 and are left to the reader.

b. Second Method. To obtain a safety-test algorithm that works in polynomial time we replace for each tree template tt its set of graphs INSTANCE-SET(tt) by a single graph INSTANCE_{tt}. A similar approach is followed for the sets of graphs CEFOL-SET(tt, k) which are replaced by the single graphs $\text{CEFOL}_{tt,k}$.

Definition 5.13. For each tree template tt with leaves $Ltt_1, Ltt_2, \ldots, Ltt_n$,

$$\text{INSTANCE}_{tt} = TG_{tt} - 0[IS_{Ltt_1}, IS_{Ltt_2}, \ldots, IS_{Ltt_n}]. \quad \square$$

Definition 5.14. For each tree template tt with root Rtt and leaves Ltt_1, $Ltt_2, \ldots, Ltt_n$ and integer $k (1 \leqq k \leqq n)$,

$$\text{CEFOL}_{tt,k} = TG_{tt} - k[SI_{Rtt}, IS_{Ltt_1}, \ldots, IS_{Ltt_{k-1}}, IS_{Ltt_{k+1}}, \ldots, IS_{Ltt_n}]. \quad \square$$

To express the time complexity of the algorithms to compute the instance graphs and cefol graphs we use the same cost values for primitive operations as at the end of Sect. 5.2. Furthermore we introduce the following parameters:

$|L|$ the maximum number of leaves in a tree template;
$|T|$ the number of tree transformation rules;
$|X|$ the maximum number of attributes of a grammar symbol.

Assuming the availability of the template graph TG_{tt} for each tree template tt and the graphs IS_X and SI_X for each $X \in V$, the computation of the graphs INSTANCE_{tt} for each tree template tt takes time $O(|T| \, |L| \, |X|^2)$. The time complexity of an algorithm to compute for each tree template tt and for each excluded leaf k the graphs $\text{CEFOL}_{tt,k}$ is $O(|T| \, |L|^2 \, |X|^2)$.

5.5 The Safety Test

In this Section algorithms are presented which investigate for an attribute grammar AG and an associated set TR of conditional tree transformation rules whether TR is safe with respect to a left-to-right pass.

These algorithms construct for each pair of tree transformation rules q: dir_q, itt_q, ott_q, cond_q, eval_q) and r: (dir_r, itt_r, ott_r, cond_r, eval_r) and for each situation as pictured in Fig. 2, the labeled graphs dg(ott_q, itt_r) expressing the

dependencies between attribute instances in EVAL(q) and attribute instances in COR(r). For each graph dg(ott$_q$, itt$_r$) the criteria of Theorem 4.1 are checked.

Two methods are discussed to construct the graphs dg(ott$_q$, itt$_r$). Both methods make use of cousin graphs, ancestor-descendant graphs, instance graphs and cefol graphs. The first method selects its graphs from sets of different graphs. The second method uses graphs showing a mixture of the dependencies exhibited by the different graphs included in a set.

a. First Method. In Algorithm 5.10 the function "test dependencies" investigates whether the dependencies between attribute instances in EVAL(q) and attribute instances in COR(r) allow the correct recomputation of the latter between the successive non-overlapping applications of q and r.

Algorithm 5.10. Computation of attribute dependencies which indicate whether or not a set TR of conditional tree transformation rules is safe with respect to an attribute grammar AG.

Input: attribute grammar AG; set TR of tree transformation rules;
sets COUSIN-SET(X, Y) and ANC-DESC-SET(X, Y) for all $X, Y \in V_N$;
sets INSTANCE-SET(tt) and CEFOL-SET(tt, k) for each input or output template tt and $1 \leq k \leq n$, where n is the number of leaves of tt.

Output: value of safe, i.e., indication whether TR is safe or not.

Algorithm:

```
const OK = true;
type dg-of-ott-and-itt = ... ; {a directed graph whose vertices are associated
                                 with attribute instances of tree template ott and
                                 itt; to each arc the label L or L̄ is assigned}
var safe: boolean;
function test dependencies (dg(ott_q, itt_r)*: dg-of-ott-and-itt; q, r: TR): boo-
        lean;
    begin
        test dependencies := OK;
        for each pair of attribute instances (a, ott_q, X) in EVAL(q)
            and (b, itt_r, Y) in COR(r), such that
            dg(ott_q, itt_r)* includes an arc ((a, ott_q, X), (b, itt_r, Y))

        do {check the criteria of Theorem 4.1}
            if ((a, ott_q, X), (b, itt_r, Y)) is labeled L̄
                or X is a leaf of ott_q and dir_q = up
                or Y is a leaf of itt_r and dir_r = down
            then test dependencies := not OK
            fi
        od
    end {of test dependencies};
safe := OK;
for each ordered pair q, r ∈ TR
    where q = (dir_q, itt_q, ott_q, cond_q, eval_q) and
          r = (dir_r, itt_r, ott_r, cond_r, eval_r)
```

```
do {let Rott_q and Ritt_r be the root of ott_q and itt_r, respectively}
   {test the case that Rott_q and Ritt_r are cousins; see Fig. 9}
   for all graphs instance(ott_q) from INSTANCE-SET(ott_q)
   do for all graphs instance(itt_r) from INSTANCE-SET(itt_r)
      do for all graphs cousin(Rott_q, Ritt_r) from
               COUSIN-SET(Rott_q, Ritt_r)
         do construct the graph dg(ott_q, itt_r) composed of the subgraphs
               instance(ott_q) and instance(itt_r);
               add labeled arcs
               ((a, ott_q, Rott_q), (b, ott_q, Rott_q)),
               ((a, ott_q, Rott_q), (d, itt_r, Ritt_r)),
               ((c, itt_r, Ritt_r), (d, itt_r, Ritt_r)),
               and ((c, itt_r, Ritt_r), (b, ott_q, Rott_q))
               if and only if there is a corresponding arc in
               cousin(Rott_q, Ritt_r);
               construct the transitive closure dg(ott_q, itt_r)*;
               safe:=safe and test dependencies (dg(ott_q, itt_r)*, q, r)
         od
      od
   od;
   {test the case where Ritt_r is a descendant of Rott_q; see Fig. 11}
   if dir_q = down
   then {let n be the number of leaves of ott_q}
      for 1 ≦ k ≦ n
      do for all graphs cefol(ott_q, k) from CEFOL-SET(ott_q, k)
         do for all graphs instance(itt_r) from INSTANCE-SET(itt_r)
            do {let Lott_qk be the k-th leaf of ott_q}
               for all graphs anc-desc(Lott_q,k, Ritt_r)
                  from ANC-DESC-SET(Lott_qk, Ritt_r)
               do construct the graph dg(ott_q, itt_r) composed of the
                  subgraphs cefol(ott_q, k) and instance(itt_r);
                  add labeled arcs
                  ((a, ott_q, Lott_qk), (b, ott_q, Lott_qk)),
                  ((a, ott_q, Lott_qk), (d, itt_r, Ritt_r))
                  ((c, itt_r, Ritt_r), (d, itt_r, Ritt_r))
                  and ((c, itt_r, Ritt_r), (b, ott_q, Lott_qk))
                  if and only if there is a corresponding arc in
                  anc-desc(Lott_qk, Ritt_r);
                  construct the transitive closure dg(ott_q, itt_r)*;
                  safe:=safe and
                           test dependencies (dg(ott_q, itt_r)*, q, r)
               od
            od
         od
      od
   fi;
   {test the case where Ritt_r is an ancestor of Rott_q}
```

if $\text{dir}_r = \text{up}$
then {as above, but exchange ott_q and itt_r,
except in $\text{dg}(\text{ott}_q, \text{itt}_r)$}
fi
od; □

From the fact that the number of graphs in the sets COUSIN-SET(X, Y) and ANC-DESC-SET(X, Y) may be exponential in the size of the grammar it follows that the time complexity of Algorithm 5.10 is exponential. Experiments will be necessary to determine whether the method applied in Algorithm 5.10 is feasible for practical attribute grammars.

Observe that for Algorithm 5.11 we need sets of graphs COUSIN-SET(X, Y) and ANC-DESC-SET(X, Y) in V_N only.

b. Second Method. The safety-test algorithm using single cousin, ancestor-descendant, instance and cefol graphs instead of sets of graphs is just a simple variation of Algorithm 5.10. The development of this algorithm is left to the reader. Observe that such an algorithm only gives a *sufficient* criterion for safety.

Using the same parameters and the same primitive operations with their associated cost values as at the end of Sect. 5.4, the time complexity of this algorithm is $O(|T|^2|L|^4|X|^3)$.

5.6 Comparison of Methods

Let Algorithm 5.11 be the variant of Algorithm 5.10, using single graphs instead of sets of graphs. Investigation of practical examples has to show whether Algorithm 5.10 really leads to better results and is preferable to Algorithm 5.11.

Let (AG, TR) denote an attribute grammar AG with an associated set TR of conditional tree transformation rules. Obviously, the class of pairs (AG, TR) accepted by Algorithm 5.11 is included in the class of pairs accepted by Algorithm 5.10. The following example shows that the inclusion is proper.

Example 5.1. Consider attribute grammar AG1 with $V_N = \{Z, A, B, C\}$, $V_T = \{a, b, c, d, e, f\}$ and attributes

$I(Z) = \emptyset$	$I(A) = \{r\}$	$I(B) = \{t, u\}$	$I(C) = \{x\}$
$S(Z) = \{\text{result}\}$	$S(A) = \{s\}$	$S(B) = \{v, w\}$	$S(C) = \{y\}$.

Figure 12 pictures the productions with associated attribute occurrences and dependencies.

Let TR consist of the following tree transformation rules.

trans1: **transform up** ⟨A, a⟩
cond condition(r **of** A) **into** ⟨A, b⟩ **eval** s **of** A := ...
end;

trans2: **transform down** ⟨C, e⟩
cond condition(x **of** C) **into** ⟨C, f⟩ **eval** ...
end.

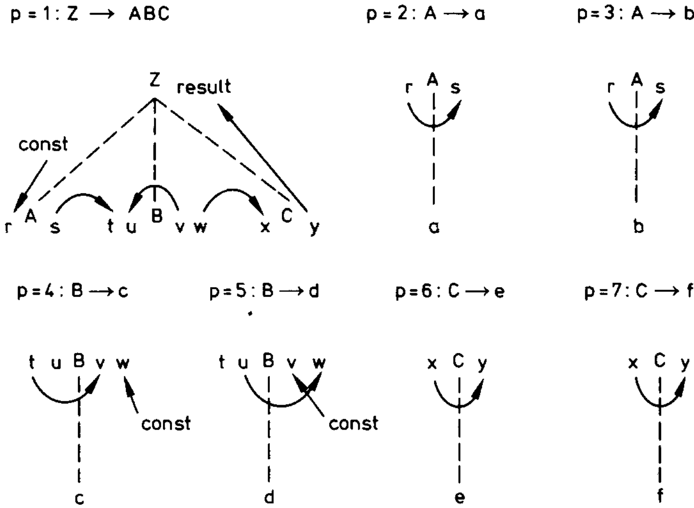

Fig. 12. Productions and attribute dependencies of attribute grammar AG1

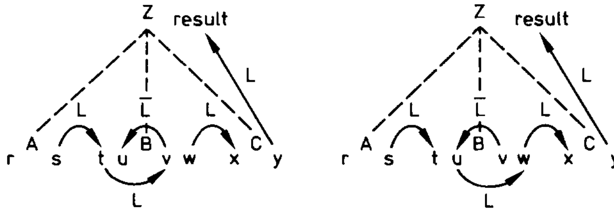

Fig. 13. Graphs DG_1-(1, 3) $[D_0, D_2]$ for $D_0 \in$SI-SET(Z) and $D_2 \in$IS-SET(B)

Algorithm 5.10 needs the set COUSIN-SET(A, C) which is equal to the set BROTHER-SET(A, C). The latter is composed of the graphs $DG_1-(1, 3)^*[D_{01}, D_{21}]$ and $DG_1-(1, 3)^*[D_{01}, D_{22}]$, where D_{01} is the single graph from SI-SET(Z) and D_{21} and D_{22} are the graphs from IS-SET(B). D_{01} is the graph with vertex result **of** Z and no arcs. D_{21} and D_{22} are found by considering productions 4 and 5. Figure 13 pictures the graphs $DG_1-(1, 3)$ $[D_{01}, D_{21}]$ and $DG_1-(1, 3)$ $[D_{01}, D_{22}]$.

COUSIN-SET(A, C) consists of a single graph with no arcs. Hence, Algorithm 5.10 concludes that it is allowed to apply transformation rules trans 1 and trans 2 during a single left-to-right pass.

Observe that the well known pass oriented evaluation strategies [1, 2, 5, 14] need two evaluation passes, whereas the transformations can be performed safely during a single pass.

Replacing sets of graphs by single graphs results in the graph $COUSIN_{A,C}$ which is equal to $DG_1-(1, 3)^*[SI_Z, IS_B]$. Figure 14 pictures the graph $DG_1-(1, 3)$ $[SI_Z, IS_B]$.

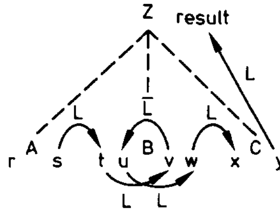

Fig. 14. Graph DG_1-(1, 3) $[SI_Z, IS_B]$

$COUSIN_{A,C}$ exhibits a dependency labeled $\bar{L}$ between attribute instances s **of** A of ott_1 and x **of** C of itt_2. Hence, from its pessimistic point of view, Algorithm 5.11 concludes that is not allowed to apply both transformations during a left-to-right pass.

6. An Example: Constant Propagation and Dead Code Elimination

The following example describes optimizing tree transformations for a small grammar including assignment, conditional, while and compound statements. The example is borrowed from [25], where it is shown how global data flow information can be collected, used in determining the applicability of optimizing tree transformations, and can be updated after invalidation of the flow information by tree transformations. The optimization algorithm described in [25] operates on abstract syntax trees, whereas the variant presented in this paper is defined in terms of concrete derivation trees.

6.1 Collection of Data Flow Information

The grammar specifying the collection of data flow information has the following attributes. Associated with each statement is a synthesized attribute mod of type set-of-ident = **set of** 1 .. max, where max is the maximal number of identifiers allowed in any program to be compiled. Attribute mod includes the identifier number of each variable which may be *mod*ified by the statement concerned. 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) of type pool are introduced. Type pool is declared to be a list of (idno, val) pairs, where idno is the number of an identifier and val its associated value. Inherited attribute *i*-pool, associated with a statement, contains the variables which are known to have the same value each time when starting the execution of the statement. Synthesized attribute *s*-pool, associated with a statement, includes the variables which are known to have the same value each time when control passes through the end of the statement.

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 existing. If it is uncertain whether the right part is a constant expression, then the pair with first component idno (if existing) is deleted from the pool of available constant variables.

When leaving a conditional statement an *s*-pool has to be returned including those (idno, val) pairs that occur identically in both the *s*-pool of the then part and the else part.

When entering a while statement, all variables assigned within the while statement have to be deleted from its associated *i*-pool.

Associated with each expression are synthesized attributes isconst of type boolean and val of type integer. Attribute isconst indicates whether the expression is known to have a constant value. For isconst = true, val denotes the associated value, otherwise the value is undefined.

Finally, attributes idno and val, associated with terminal symbols ident and const, respectively, are set by the scanner. Both attributes are of type integer.

Each production of the grammar is 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 AG2 describing the collection of data flow information is defined as follows.

Attribute Grammar AG2:

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

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

start symbol: program.

attribute types:

```
const max = ... {maximal number of identifiers};
      undefined = ... {any integer};
      empty-pool = nil;
type  set-of-ident = set of 1 .. max;
      pool = ↑pool-entry;
      pool-entry = record
                      idno: integer,
                      val:  integer,
                      next: pool
                   end.
```

semantic attributes:

```
idno:     integer, syn of ident;
val:      integer, syn of const, expr;
isconst:  boolean, syn of expr;
```

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: **syn of** compound, stats, stat, assignment, condstat, whilestat.

functions:

function initialize-mod-with (idno: integer) **delivers** set-of-ident:
  begin initialize-mod-with:=[idno] **end**;
function insert (idno, val: integer) into: (*p*: pool) **delivers** pool:
  begin {inserts a new pair (idno, val) into the pool *p*,
    replacing a pair with the same idno, if existing}
  end;
function delete (idno: integer) from: (*p*: pool) **delivers** pool:
  begin {delete the pair with first component idno, if existing} **end**;
function intersect (*p*1, *p*2: pool) **delivers** pool:
  begin {returns a pool containing those pairs that occur identically in both input pools *p*1 and *p*2}
  end;
function delete-all-identifiers-in (mod: set-of-ident) from: (*p*: pool) **delivers** pool:
  begin {eliminates all pairs (idno, val) from pool *p* for which idno in mod}
  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 isonst of expr
            then insert (idno of ident, val of expr) 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 constat:=mod of stats [1]+mod of stats [2];
          i-pool of cond:=i-pool of condstat;
          i-pool of stats [1]:=i-pool of condstat;
          i-pool of stats [2]:=i-pool of condstat;
          s-pool of condstat:=intersect ( s-pool of stats [1],
                                          s-pool of stats [2])
        ]
(11) whilestat → while cond do stats od.
        [ mod of whilestat:=mod of stats;
          i-pool of cond:=delete-all-identifiers-in (mod of stats) from:
                                                        (i-pool of whilestat);
          i-pool of stats:=delete-all-identifiers-in (mod of stats) from:
                                                        (i-pool of whilestat);
          s-pool of whilestat:=delete-all-identifiers-in (mod of stats) from:
                                                        (i-pool of whilestat)
        ]
(12) cond → expr [1]=expr [2].
        [ i-pool of expr [1]:=i-pool of cond;
          i-pool of expr [2]:=i-pool of cond
        ]
(13) expr [1] → expr [2]+expr [3].
        [ isconst of expr [1]:=false;
          val of expr [1]:=undefined;
          i-pool of expr [2]:=i-pool of expr [1];
          i-pool of expr [3]:=i-pool of expr [1]
        ]
(14) expr → ident.
        [ isconst of expr:=false;
          val of expr:=undefined
        ]
(15) expr → const.
        [ isconst of expr:=true;
          val of expr:=val of const
        ]
```

Production rule (13) causes the grammar to be ambiguous. However, the requirement that the plus operator be left associative is sufficient to disambiguate the grammar.

The simple multi-pass attribute evaluation strategy [1, 2, 5, 14] requires that with each attribute it is possible to associate a fixed pass number such that the evaluation of all instances of that attribute in any derivation tree of

the grammar can be performed during that pass. From the restriction of this strategy it follows [1, 2] that at least two left-to-right passes are needed to evaluate all attribute instances within any derivation tree of attribute grammar AG2 (See production (11), where *i*-pool **of** stats depends on mod **of** stats). The instances of indo **of** ident and val **of** const are defined by the parser. Restricting the number of passes to two, the pass numbers for the remaining attributes are as follows.

attribute	*pass number*
val **of** expr	1 or 2
isconst **of** expr	1 or 2
mod **of** any	1
i-pool **of** any	2
s-pool **of** any	2

6.2 Transformation Rules

First we define two functions to be applied in the enabling conditions and the re-evaluation rules of conditional tree transformation rules defined for the benefit of constant propagation.

functions:

function element (idno: integer) 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: integer) in: (*p*: pool) **delivers** integer:
  begin {returns the value belonging to idno in pool *p*} **end**.

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

transformation rules:

trans1: **transform up** ⟨expr, ident⟩
  cond element (idno **of** ident) in: (*i*-pool **of** expr)
    into ⟨expr, const⟩
      eval val **of** const := value-of (idno **of** ident) in: (*i*-pool **of** expr);
        isconst **of** expr := true;
        val **of** expr := value-of (idno **of** ident) in: (*i*-pool **of** expr)
  end;
trans2: **transform up** ⟨expr, ⟨expr, const [1]⟩, +, ⟨expr, const [2]⟩⟩
  into ⟨expr, const⟩
    eval val **of** const := val **of** const [1] + val **of** const [2]
      isconst **of** expr := true;
      val **of** expr := val **of** const [1] + val **of** const [2]
  end.

Transformation rule trans1 may also be applied during a downward move.

Both the application of Algorithm 5.10 (first method) and its variant (second method) lead to the conclusion that tree transformations trans 1 and trans 2 can be applied safely (i.e., {trans 1, trans 2} is safe) during a single left-to-right pass over any derivation tree of attribute grammar AG2. Observe that not a single instance of attribute mod is invalidated either by trans 1 or by trans 2, but that instances of *i*-pool and *s*-pool may have to be recomputed. This allows a safe performance of tree transformations and attribute re-evaluations during a single left-to-right pass, although the initial computation of the attributes required two passes.

Observe that rule (14) of AG2 could have been defined as follows.

```
(14) expr → ident.
        [ isconst of expr:=element (idno of ident) in: (i-pool of expr);
          val of expr:=if element (idno of ident) in: (i-pool of expr)
                       then value-of (idno of ident) in: (i-pool of expr)
                       else undefined
                       fi
        ]
```

These new semantic rules allow the replacement of trans 1 by:

```
trans 1: transform up ⟨expr, ident⟩
             cond element (idno of ident) in: (i-pool of expr)
                into ⟨expr, const⟩
                   eval val of const:=value-of (idno of ident) in:
                                                              (i-pool of expr)
         end.
```

For this new version of trans 1 no recomputations are needed during the continuation of the transformation pass (i.e., after the application of trans 1), because EVAL(trans 1) is empty. Notice, that also rule (13) and trans 2 can be changed such that EVAL(trans 2) is empty. These changes are natural, but make our example less interesting.

Also observe that the example allows the overlapping application of tree transformation rules, since ott_1 and ott_2 (of trans 1 and trans 2, respectively) fit into itt_2 (of trans 2). Although the problem of overlapping tree transformations is not included in our theory (this is a topic for further research), for trans 1 and trans 2 it is easily checked that they can be applied safely after each other even when they are involved in overlap.

Figure 15 shows that, in case of overlap, no re-evaluations are needed in between the application of trans 1 or trans 2 and the application of trans 2. Moreover, given a sequence of overlapping applications of tree transformation rules, no derivation tree will include a dependency path starting from an EVAL attribute instance of a transformation applied before the sequence, leading to a COR attribute instance of a transformation to be applied after the sequence and running through the input or the output template of a transformation forming part of the sequence.

Now, we add another tree transformation rule which specifies the replacement of a conditional statement, in case of a compile-time evaluable condition, by its then part or its else part.

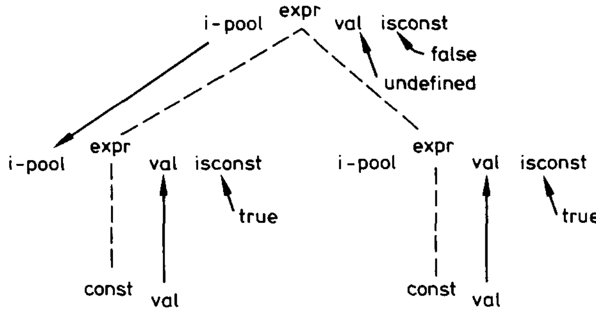

Fig. 15. Dependencies between attribute instances involved in overlapping tree transformations

```
trans3:  transform up ⟨stat,
                     ⟨condstat, if, ⟨cond, ⟨expr, const [1]⟩, =,
                      ⟨expr, const [2]⟩⟩, then, stats [1], else, stats [2], fi
                     ⟩
                    ⟩
      cond val of const [1] = val of const [2]
         into ⟨stat, ⟨compound, begin, stats [1], end⟩⟩
            eval mod of compound := mod of stats [1];
                 mod of stat := mod of stats [1];
                 i-pool of compound := i-pool of stat;
                 s-pool of compound := s-pool of stats [1];
                 s-pool of stat = s-pool of stats [1]
      cond val of const [1] ≠ val of const [2]
         into ⟨stat, ⟨compound, begin, stats [2], end⟩⟩
            eval mod of compound := mod of stats [2];
                 mod of stat := mod of stats [2];
                 i-pool of compound := i-pool of stat;
                 s-pool of compound := s-pool of stats [2];
                 s-pool of stat := s-pool of stats [2]
    end.
```

In a similar manner a tree transformation rule can be defined specifying the conditional replacement of a while statement by a loop forever or a no-operation.

Now, after the addition of rule trans3, both Algorithm 5.10 and its variant indicate that the specified transformation rules cannot be applied safely during a single pass over all derivation trees of attribute grammar AG2. This is caused by the attribute dependencies of the while statement, as illustrated in Fig. 16, where terminal symbols are deleted. The replacement of the conditional statement by its then part or its else part may cause instances of attribute mod to get different values and this, in its turn, may require the recomputation of instances of *i*-pool and *s*-pool, needed for additional transformations within stats [2]. The situation pictured in Fig. 16 shows that the recomputation, during a single pass, of instances of both mod and *i*-pool and *s*-pool is generally impossible.

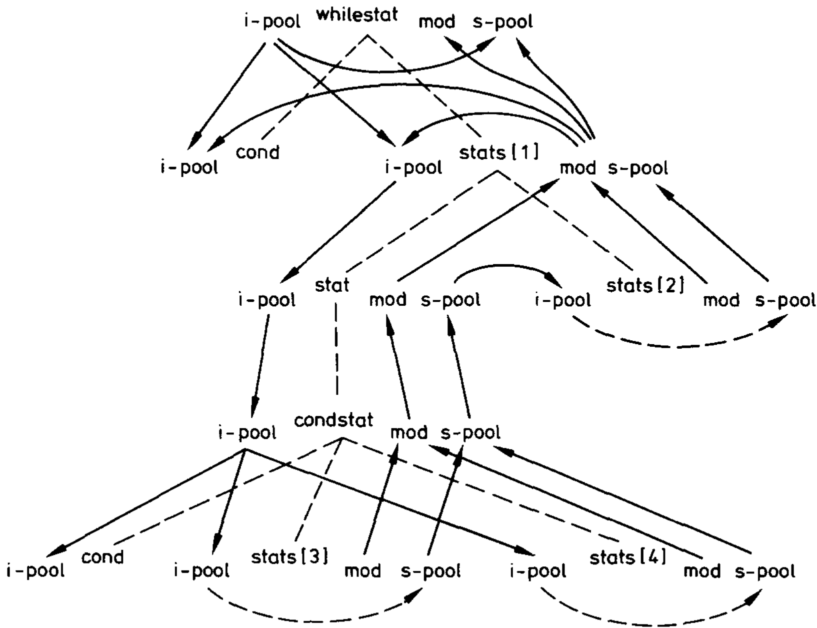

Fig. 16. Indirect dependencies between attribute instances of statements of a while body

Observe that itt_3 (of trans3) may cover ott_1 and ott_2 (of trans1 and trans2, respectively). We pay no further attention to this problem, since {trans1, trans2, trans3} already turned out to be unsafe for non-overlapping tree transformations.

6.3 Iterative Application of a Transformation Pass

Up to now we assumed one single value to be correct for each attribute instance within a derivation tree. Now we allow a certain set of values to be correct for each instance of mod, *i*-pool and *s*-pool.

A correct value of mod of a statement includes at least the identifier numbers of all variables assigned by the statement. It may include more identifier numbers, but the best value includes only the variables assigned within the statement. A correct value of *i*-pool and *s*-pool is a list containing a subset of the (idno, val) pairs of variables which are known to be constant at the associated point in the program. The best value is the list including the (idno, val) pairs of all constant variables.

In [9, 10] the correct values are called safe, whereas the best values are called consistent.

Using safeness as the new definition of correctness we may conclude that after the application of transformation rule trans3 and during the continuation of the transformation pass it is not always possible to compute the best value for the instances of *i*-pool and *s*-pool, although their values are always safe.

This means that all the transformations performed during the transformation pass are correct, but in interrupt of the transformation pass in order to make extra tree walks for re-evaluation purposes might have disclosed further opportunities for optimization [9, 10].

If we want to avoid extra re-evaluation passes after each tree transformation, then, for this example, the following approach might be considered. First, recognize that the second evaluation pass, during which the instances of *i*-pool and *s*-pool are computed, may be combined with the transformation pass, since all the necessary attribute values are available in time. If the initial values of the instances of isconst, val (**of** expr) and mod are computed during the first pass, then, during the second pass, the values of the instances of *i*-pool and *s*-pool can be computed, using the currently available values of isconst, idno, val and mod. Moreover, during this pass it is possible to perform transformations using the currently available values of instances of *i*-pool, idno and val. Observe that during the second pass it is also possible to recompute the instances of isconst, idno, val and mod. This suggests and allows the repetition of the second pass until no more changes take place.

The following program shows an example where the repetition of the transformation pass leads to further improvements. Assume that besides a transformation rule for the conditional statement a similar rule for the while statement has been defined.

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

During the first execution of the transformation pass the conditional statement is replaced by its then part. During the second execution the while statement is replaced by the construction

forever do begin $d := 1$ **end od**

which, of course should result in a warning.

As in [4] it can be proved for attribute grammar AG2 that, for any program which contains W while statements and C conditional statements, at most $W + C + 1$ executions of the transformation pass are needed to do all possible constant folding and to eliminate all dead code.

Acknowledgements. I would like to thank Tartan Laboratories Inc. for the opportunity to spend my sabbatical in Pittsburgh, PA, where I prepared a preliminary version of this paper. Special thanks go to John R. Nestor who gave me the topic, William L. Scherlis for discussing the results of the research, and to Suzanne M. Broughton for helping me to prepare the first version of this paper. Fruitful discussions with Joost Engelfriet contributed to the precision and clarity of the final version. Thanks are also due to Thérèse ter Heide for her help in preparing the final version of the manuscript.

References

1. Alblas, H.: A characterization of attribute evaluation in passes. Acta Inf. **16**, 427–464 (1981)
2. Alblas, H.: Finding minimal pass sequences for attribute grammars. SIAM J. Comput. **14**, 889–914 (1985)
3. Alblas, H.: Incremental simple multi-pass attribute evaluation. Proc. NGI/SION Symposium 1986, 319–342 (1986)
4. Babich, W.A., Jazayeri, M.: The method of attributes for data flow analysis, Part I. Exhaustive analysis. Acta Inf. **10**, 245–264 (1978)
5. Bochmann, G.V.: Semantic evaluation from left to right. Commun. ACM **19**, 55–62 (1976)
6. Demers, A., Reps, T., Teitelbaum, T.: Incremental evaluation for attribute grammars with application to syntax-directed editors. Proc. Eighth ACM Symp. Principles Programm. Lang., pp. 105–116 (1981)
7. DeRemer, F.L.: Transformational grammars. In: Compiler construction: An Advanced Course. Bauer, F.L., Eickel, J. (eds), Lect. Notes Comput. Sci., Vol. 21, pp. 121–145. Berlin-Heidelberg-New York: Springer 1974
8. Engelfriet, J.: Attribute grammars: Attribute Evaluation Methods. In: Methods and Tools for Compiler Construction, pp. 103–138. Cambridge University Press 1984
9. Ganzinger, H., Giegerich, R.: A truly generative semantics directed compiler generator. In: Proc. SIGPLAN 1982 Symposium on Compiler Construction. SIGPLAN Notices **17**, 6 172–184 (1982)
10. Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximative semantics with respect to program transformations. In: Informatik-Fachberichte, Vol. 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, Vol. 34, pp. 125–142. Berlin-Heidelberg-New York: Springer 1980
12. Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attribute grammars. Commun. ACM **18**, 697–706 (1975)
13. Jazayeri, M.: A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars. J. ACM **28**, 715–720 (1981)
14. Jazayeri, M., Walter, K.G.: Alternating semantic evaluator. Proc. ACM 1975 Annual Conference, 230–234 (1975)
15. Kastens, U.: Ordered attribute grammars. Acta Inf. **13**, 229–256 (1980)
16. Kennedey, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. Proc. Third ACM Symp. Principles Programm. Lang., pp. 32–49 (1976)
17. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**, 127–145 (1968)
18. Knuth, D.E.: Semantics of context-free languages: Correction. Math. Syst. Theory **5**, 95–96 (1971)
19. Möncke, U., Weisgerber, B., Wilhelm, R.: How to implement a system for manipulation of attributed trees. In: Informatik Fachberichte, Vol. 77, pp. 112–127. Berlin-Heidelberg-New York-Tokyo: Springer 1984
20. Möncke, U., Wilhelm, R.: Iterative algorithms on grammar graphs. In: Proc 8th Conference on Graphtheoretic Concepts in Computer Science, pp. 177–194. München-Wien: Hanser 1982
21. Nestor, J.R., Mishra, B., Scherlis, W.L., Wulf, W.A.: Extensions to attribute grammars. Technical Report TL 83-36, Tartan Laboratories Incorporated, 1983
22. Räihä, K.-J., Saarinen, M.: Testing attribute grammars for circularity. Acta Inf. **17**, 185–192 (1982)
23. Reps, T.: Optimal-time incremental semantic analysis for syntax directed editors. Proc. Ninth ACM Symp. Principles Programm. Lang. pp. 169–176 (1982)
24. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language based editors. ACM Trans. Programm. Lang. **5**, 449–477 (1983)
25. Wilhelm, R.: Computation and use of data flow information in optimizing compilers. Acta Inf. **12**, 209–225 (1979)
26. Yeh, D.: On incremental evaluation of ordered attribute grammars. BIT **23**, 308–320 (1983)

Received September 10, 1984/October 6, 1986