Incremental Evaluators for Ordered Macro Attribute Grammars

I. Adamski
B. Jokubeit
A. Kühnemann
H. Vogler

Department of Computer Science
Dresden University of Technology
01062 Dresden, Germany

e–mail: {adamski, jokubeit, kuehne, vogler}@orchid.inf.tu-dresden.de

Abstract

We define macro attribute grammars which are a generalization of attribute grammars. The generalization is the result of the introduction of attributes whose values can be functions. We introduce the subclass of ordered macro attribute grammars. For every ordered macro grammar we present an incremental attribute evaluator, which is based on a term representation for values of attribute instances. Using a graph representation for values of attribute instances, we show, how the efficiency of the incremental evaluators can be improved.

1 Introduction

Attribute grammars (for short: ags) were introduced by D. E. Knuth in [Knu68] to assign "meaning" to the words of context-free languages. Since 1968 much research has been done to study properties of ags (cf. [Cou84], [DJL86], [DJL88], [AM91], [Paa95], [KV97]). Special fields of interest were the expressive power of the formal model (cf. [Eng80], [Fül81], [EF89], [FHVV93], [KV94], [Küh97]) and the property of circularity. Subclasses of ags were created for the development of efficient algorithms for attribute evaluation, e.g., *S-ag* and *L-ag* (cf. [LRS74]), *multi-pass ags*, *multi-sweep ags*, and *multi-visit ags* (cf. [EF89], [AM91]), *absolutely noncircular ags* (cf. [KW76]), *ordered ags* (cf. [Kas80]), and *partionable ordered ags* (cf. [WG84]). A number of extensions of ags have been proposed, e.g., *attribute-coupled grammars* (cf. [GG84], [Gie88]) and *higher-order ags* (cf. [KSV89]). In *attribute-coupled grammars* and *higher-order ags* the value of an attribute can be a derivation tree, whose attribute values are subsequently evaluated. In [Mad97], a metalanguage for the specification of the static semantic of programming languages is introduced. The metalanguage, which is called *Analysis Definition Language*, enriches the classical ag notion by allowing synthesized attributes to be defined as first-order functions, relations, or objects. These extensions increase the expressive power of the specification language "attribute grammar" in different ways.

In this paper we consider another extension of ags, called *macro attribute grammars* (for short: mags). This extension is inspired by the formal model of *macro attributed tree transducers* (cf. [KV94], [FV98]). Like an ag, an mag specifies the meaning of a derivation tree t as a value of a designated attribute α_0 at the root of t. The attribute α_0, called *meaning attribute*, owns a basic semantic value. The other attributes, which are associated with the nodes of t, are *functions* with parameters over basic semantic values. This gives the possibility to specify and maintain context information in parameter positions. As usual, attributes are partitioned into *synthesized attributes* and *inherited attributes*. The circularity test for mags is the same as the circularity test for ags. In [Jok96] *ordered mags* are derived from ordered ags. Based on the idea of the evaluator for ordered ags, in [Jok96] an iterative tree walking evaluator for ordered mags has been developed.

Here we investigate incremental attribute evaluation algorithms for ordered mags. Such algorithms are applicable in language based editors in which the derivation tree is subjected to modifications. The main operation in a language based editor is the *subtree substitution*, which implements the edit operation. After a subtree substitution some attributes of the derivation tree are retained — either changed or equal — and some are new. In consequence the values of some attributes have to be *recomputed*. The application of the iterative evaluator would solve this problem by recomputation of *all* attributes. But this is not as efficient as it could be.

By focusing on those attributes, which have a new or a changed value, we obtain an optimized algorithm for the recomputation. This technique belongs to a general class of optimization techniques, called *incremental computation*. An overview of incremental computation is given in [RR93]. Using the terminology introduced in [RR93] our problem belongs to the class of *data structure update problems*, in particular to the problems of *incremental updating of annotations on graphs and trees* which are solved by the method *selective recomputation* or by the method *differential updating*. In [RR93], the method *selective recomputation* is characterized as follows:

> "*In selective re-computation, annotation values that are independent of changed data are never recomputed. Annotation values that are dependent on changed data are recomputed, but after each new annotation value is obtained, the old and new annotation values may be compared in order to help determine what further re-computations must be performed.*"

Note, that in our case the *changed data* are modified derivation trees and the *annotation values* are the attribute values of the derivation trees.

Another characterization of incremental methods is given in [vdM94], where incremental methods are classified as *coarse-grain* or *fine-grain* due to its view on the attribute values. The selective recomputation is a coarse-grain incremental method, because the method is incremental due to the selection of attributes, but it is not incremental due to the evaluation of the attribute values itself. Note that the recomputation of complex attribute values such as tables, sets, and trees often triggers the recomputation in more places than intuition tells. A fine-grain method for attribute evaluation requires the *combination* of selective recomputation with an incremental computation of the attribute values themselves. Some mechanisms for the evaluation of complex attributes can be found in [RMT86] and in [HT86].

An incremental evaluator for *ordered ags* based on selective recomputation was developed and implemented by Reps, Teitelbaum, and Demers (cf. [DRT83], [RT89]). An efficient incremental evaluator for *ordered higher-order ags*, which applies nested visit functions to traverse and decorate a derivation tree, is described in [KSV89]. Based on the original algorithm for ordered ags we will first develop an incremental evaluator for *ordered mags*, which is coarse-grain (cf. [Jok96] for a version of this incremental evaluator). Our coarse-grain evaluator serves as starting point to develop a fine-grain incremental evaluator for ordered mags.

To get an idea how a fine-grain method can be achieved, let us explain a bit more about the incremental recomputation of attributes in ordered mags. Recall, that the values of attributes are functions and thus, the evaluation requires the representation of the values as *terms*. In consequence, we have to use a technique of *incremental term rewriting* for the implementation of a fine-grain method of attribute evaluation.

In [Fie93] an incremental term rewriting method for arbitrary left linear term rewriting systems has been proposed. However, this universal technique is too powerful for our special purposes, because it allows arbitrary subterms to be rewritten by arbitrary substituents, where the replaceable subterms can be arbitrarily nested. A simpler incremental term rewriting technique has been developed for well-presented primitive recursive program schemes in [vdM94]. The incremental evaluation of well-presented primitive recursive program schemes is realized by an incremental evaluator for absolutely noncircular ags, which leads to an efficient method. The analogy between well-presented primitive recursive program schemes and absolutely noncircular ags has been proved in [CF82] (also cf. [FV98]).

To obtain a *fine-grain incremental attribute evaluator for ordered mags*, let us have a closer look at the structure of the term rewriting system induced by our computational model. With every node x of a derivation tree t a set of attribute instances is associated. Let $\langle a, x\rangle$ be such an attribute instance, which is associated with the node x. The value of $\langle a, x\rangle$ is specified by a *semantic equation* of the following form:

$$\langle a, x\rangle(y_1, \ldots, y_k) = \zeta,$$

where ζ is a term over function symbols of the semantic domain, attribute instances, and parameters $y_1, \ldots, y_k$. The attribute instances occurring in ζ are only those attribute instances, which are associated with x itself or with its direct successors or with its direct predecessor. In this sense, the dependencies between the attribute instances are *local.* Intuitively, to compute the term, which represents the value of $\langle a, x\rangle$, we have to substitute every attribute instance in ζ by the term, which represents its value.

In this paper, we represent ζ as a graph. Then we are able to realize the computation of the value of $\langle a, x\rangle$ by *graph substitution.*

Example 1.1 (Updating graphs). Let t be a derivation tree which has at least the three nodes 1211, 1212, and 121 where a node is represented by its Dewey-notation. Assume that the values of the three attribute instances $\langle v, 1211\rangle$, $\langle v, 1212\rangle$, and $\langle v, 121\rangle$ of the derivation tree t are specified by the following semantic equations

$$\begin{aligned} \langle v, 1211\rangle(y) &= y, \\ \langle v, 1212\rangle(y) &= s(s(y)), \text{ and} \\ \langle v, 121\rangle(y) &= \langle v, 1211\rangle(\langle v, 1212\rangle(y)). \qquad (*) \end{aligned}$$

Thus, interpreting s as the successor function over the natural numbers $I\!N$, the values of the attribute instances are the functions $val_t(\langle v, 1211\rangle)$: $I\!N \to I\!N$, $val_t(\langle v, 1212\rangle)$: $I\!N \to I\!N$, and $val_t(\langle v, 121\rangle)$: $I\!N \to I\!N$, respectively, which are defined as follows: For every $n \in I\!N$,

$$\begin{aligned} val_t(\langle v, 1211\rangle)(n) &= n, \\ val_t(\langle v, 1212\rangle)(n) &= n + 2, \text{ and} \\ val_t(\langle v, 121\rangle)(n) &= n + 2. \end{aligned}$$

For these attribute instances, we represent their values, as well as their dependencies on other attribute instances by the hypergraphs shown in Figure 1a, which we call *extended parjungles.* Now imagine that the derivation tree t is subjected to a modification which yields t_{new} and which introduces the new semantic equation

$$\langle v, 1211\rangle(y) = s^4(y),$$

as the specification of the value of $\langle v, 1211\rangle$. Thus, the new value of $\langle v, 1211\rangle$ will be evaluated to the function $val_{t_{new}}(\langle v, 1211\rangle)$: $I\!N \to I\!N$ defined by

$$val_{t_{new}}(\langle v, 1211\rangle)(n) = n + 4, \text{ for every } n \in I\!N,$$

which is represented as $graph_{t_{new}}(\langle v, 1211\rangle)$ in Figure 1b.

As a consequence of the dependence of the instance $\langle v, 121\rangle$ on the instance $\langle v, 1211\rangle$, which is specified by the semantic equation $(*)$, the new value of $\langle v, 1211\rangle$ induces a new value of $\langle v, 121\rangle$. Instead of performing a complete recomputation of the value of $\langle v, 121\rangle$, our approach allows to obtain the new value of $\langle v, 121\rangle$ by *modifying* the graph, which represents the old value of $\langle v, 121\rangle$. Intuitively, only those subgraphs are affected

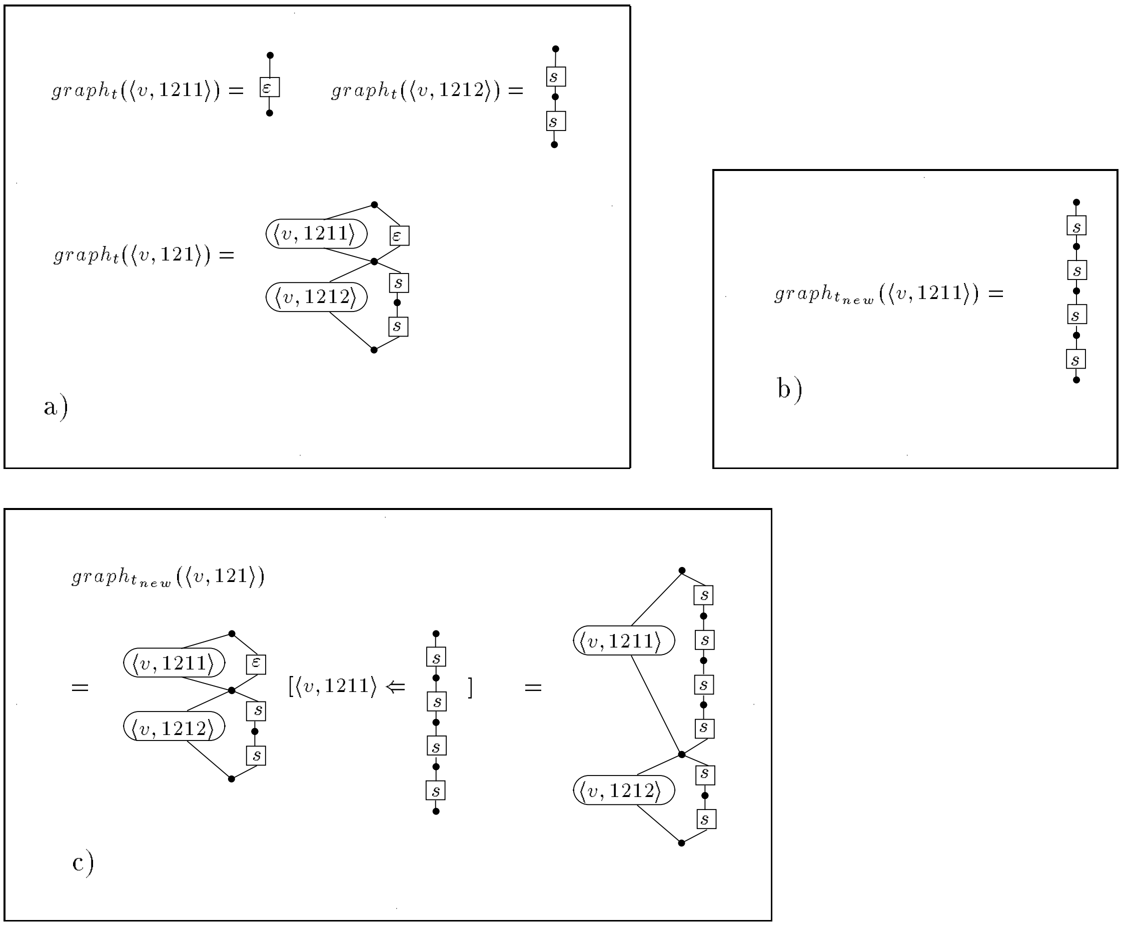


Figure 1: Selective graph substitution.

by the changed value of $\langle v, 1211\rangle$, which are the smallest subgraphs containing a *copy* of the graph of $\langle v, 1211\rangle$. To detect and to replace these subgraphs we have developed the notion of *extended parjungles* and *extended hyperedges* and a method for selective graph substitution, which we call *extended hyperedge substitution.* Intuitively, the edge in $graph_t(\langle v, 121\rangle)$, which is labeled by $\langle v, 1211\rangle$, is a "mark" for a copy of the parjungle, which represents the current value of $\langle v, 1211\rangle$. Thus, the copy of $graph_t(\langle v, 1211\rangle)$ in $graph_t(\langle v, 121\rangle)$ can be completely handled by considering all edges and vertices between the incident vertices of the edge with the label $\langle v, 1211\rangle$. The substitution step, which is necessary to update $graph_t(\langle v, 121\rangle)$, is illustrated in Figure 1c. In general, if a graph represents a function with n arguments, then the "mark" has $n+1$ external vertices . ■

The *selective recomputation* combined with the *selective graph substitution* yields a fine-grain incremental evaluator for mags, which we introduce in our paper.

The remainder of the paper is organized as follows: Section 2 presents basic notations and definitions, which will be used throughout our paper. Section 3 introduces the concept of mag. Section 4 outlines two algorithms for attribute evaluation of ordered mags in a term decoration setting. Section 5 describes the optimization of incremental attribute evaluation by graph substitution.

2 Preliminaries

We recall some basic terminology and notation, which is used throughout our paper. Besides some general notations, we will use sorted sets and sorted trees, mappings of trees into a semantic domain, i.e., interpretation functions, directed (acyclic) graphs and topological sortings of them, hypergraphs and hyperedge substitution, as well as context-free grammars and derivation trees.

2.1 General notations

The empty set is denoted by $\emptyset$. For every set M the set of all subsets of M is denoted by $\mathcal{P}(M)$, and the cardinality of M is denoted by $|M|$. For two sets M and N the difference of M and N is denoted by $M \setminus N$. The set of natural numbers including 0 is denoted by $I\!N$, the set of positive integers is denoted by $I\!N_+ = I\!N \setminus \{0\}$. For two numbers $l, u \in I\!N$ the set of all numbers $n \in I\!N$ with $l \leq n \leq u$ is denoted by $\{l, \ldots, u\}$.

For every $k \in I\!N$ and for k sets $M_1, \ldots, M_k$ the cartesian product of $M_1, \ldots, M_k$ is denoted by $M_1 \times \ldots \times M_k = \{(m_1, \ldots, m_k) \mid m_i \in M_i, 1 \leq i \leq k\}$. Every subset $R \subseteq M_1 \times \ldots \times M_k$ is called a k-ary relation over $M_1, \ldots, M_k$. For three sets M_1, M_2, and M_3 and two binary relations $R_{1,2} \subseteq M_1 \times M_2$ and $R_{2,3} \subseteq M_2 \times M_3$ the composition of $R_{1,2}$ and $R_{2,3}$ is denoted by $R_{1,2} \circ R_{2,3} = \{(e_1, e_3) \mid (e_1, e_2) \in R_{1,2}$ and $(e_2, e_3) \in R_{2,3}\}$. For every set M and for every binary relation $R \subseteq M \times M$ the transitive closure of R is denoted by R^+. The reflexive and transitive closure of R is denoted by R^*. For every subset $N \subseteq M$ the restriction $R \cap (N \times N)$ of R to the elements of N is denoted by $R\lceil N\rceil$.

For every $k \in I\!N$ and for $k+1$ sets $M_0, M_1, \ldots, M_k$ the k-ary function f over $M_1, \ldots, M_k$ into M_0 is denoted by $f : M_1 \times \ldots \times M_k \to M_0$. For k subsets $N_1 \subseteq M_1, \ldots, N_k \subseteq M_k$ the restriction of f to the elements in $N_1 \times \ldots \times N_k$ is denoted by $f|_{N_1 \times \ldots \times N_k}$. For subsets $N_i, N_i' \subseteq M_i$, with $i \in \{1, \ldots, k\}$ and $N_i \cap N_i' = \emptyset$, and for two k-ary functions $f : N_1 \times \ldots \times N_k \to M_0$ and $f' : N_1' \times \ldots \times N_k' \to M_0$, the union of f and f', denoted by $(f \cup f') : (N_1 \cup N_1') \times \ldots \times (N_k \cup N_k') \to M_0$, is defined as follows. If $(n_1, \ldots, n_k) \in N_1 \times \ldots \times N_k$, then $(f \cup f')(n_1, \ldots, n_k) = f(n_1, \ldots, n_k)$. If $(n_1, \ldots, n_k) \in N_1' \times \ldots \times N_k'$, then $(f \cup f')(n_1, \ldots, n_k) = f'(n_1, \ldots, n_k)$. Otherwise $(f \cup f')(n_1, \ldots, n_k)$ is undefined.

Let Σ be a finite set of symbols. The set of all finite words over Σ is denoted by Σ^*. The empty word is denoted by ε. For every word $w = \sigma_1 \ldots \sigma_l \in \Sigma^*$ the length l of w is denoted by $|w|$, and for every $i \in \{1, \ldots, l\}$ the ith symbol $\sigma_i \in \Sigma$ occurring in w is denoted by $w(i)$. For two words $w_1, w_2 \in \Sigma^*$ the concatenation of w_1 and w_2 is denoted by $w_1 \circ w_2$, abbreviated by $w_1 w_2$. For two subsets $L_1, L_2 \subseteq \Sigma^*$ the concatenation of L_1 and L_2 is defined by $L_1 \circ L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$, which is denoted by $L_1 L_2$, too. For a word uwv with $u, v \in \Sigma^*$ and $w \in \Sigma^* \setminus \{\varepsilon\}$ we say *w occurs in uwv*. For a word w and two lists $v_1, \ldots, v_l$ and $u_1, \ldots, u_l$ of words with $l \geq 0$, we denote by $w[v_1 := u_1, \ldots, v_l := u_l]$ the word, which is obtained by replacing every v_i occurring in w by u_i, assuming that for every $i, j \in \{1, \ldots, l\}$, v_i and v_j do not overlap in w.

2.2 Sorted sets and trees

In the following let K be a nonempty set. The elements of K are called *sorts*. The elements of $\mathrm{K}^* \times \mathrm{K}$ are called sorts, too.

Definition 2.1 (K-sorted set). A K-*sorted set* is a tuple $(\Sigma, sort)$, where Σ is a finite set of symbols, and $sort : \Sigma \to \mathrm{K}$ is a mapping which maps every symbol $\sigma \in \Sigma$ to a sort $\kappa \in \mathrm{K}$.

A symbol $\sigma \in \Sigma$ with $sort(\sigma) = \kappa$ is denoted by σ^κ, too. Σ^κ denotes the subset of Σ containing all symbols σ^κ. If the mapping *sort* of an arbitrary sorted set $(\Sigma, sort)$ is known from the context, then $(\Sigma, sort)$ is abbreviated by Σ. □

In the following let Σ be a $(\mathrm{K}^* \times \mathrm{K})$-sorted set and Y be a K-sorted set. The elements of Y are called *parameters*.

Definition 2.2 (K-sorted tree). The *set of* K-*sorted trees over* Σ *indexed by* Y, denoted by $T_\Sigma(Y)$, is the smallest K-sorted set $\bigcup_{\kappa \in \mathrm{K}} T^\kappa$ such that $Y^\kappa \subseteq T^\kappa$, and for every $\sigma \in \Sigma^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$ with $k \geq 0$ and $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$, and for every $t_1 \in T^{\kappa_1}, \ldots, t_k \in T^{\kappa_k}$ the tree $\sigma(t_1, \ldots, t_k)$ is an element of T^{κ_0}.

For every $t, s \in T_\Sigma(Y)$ the following concepts are defined: the *set of nodes of* t, denoted by $node(t)$, for every node $n \in node(t)$ the *label of* n, denoted by $label_t(n)$, the *set of subtrees of* t, denoted by $sub(t)$, the *set of parameters occurring in* t, denoted by $par(t)$, the *set of symbols of* Σ *occurring in* t, denoted by $sym(t)$, and the *substitution of a subtree of* t *at a node* n *of* t *by a tree* s , denoted by $replace(t, n, s)$.
The concepts are defined inductively on the structure of t as follows:
If $t \in Y$, then

$$
\begin{array}{lll}
node(t) & = & \{\varepsilon\} \\
label_t(n) & = & t, \text{ if } n = \varepsilon \\
sub(t) & = & \{t\} \\
par(t) & = & \{t\} \\
sym(t) & = & \emptyset \\
replace(t, n, s) & = & s, \text{ if } n = \varepsilon
\end{array}
$$

If $t = \sigma(t_1, \ldots, t_k)$ with $k \geq 0$, $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$, $\sigma \in \Sigma^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$, and $t_1^{\kappa_1}, \ldots, t_k^{\kappa_k} \in T_\Sigma(Y)$, then

$$
\begin{array}{lll}
node(t) & = & \{\varepsilon\} \cup \{iv \mid i \in \{1, \ldots, k\}, v \in node(t_i)\} \\
label_t(n) & = & \sigma, \text{ if } n = \varepsilon \\
 & = & label_{t_i}(v), \text{ if } n = iv \text{ with } i \in \{1, \ldots, k\} \text{ and } v \in node(t_i) \\
sub(t) & = & \{t\} \cup \bigcup_{i \in \{1, \ldots, k\}} sub(t_i) \\
par(t) & = & \bigcup_{i \in \{1, \ldots, k\}} par(t_i) \\
sym(t) & = & \{\sigma\} \cup \bigcup_{i \in \{1, \ldots, k\}} sym(t_i) \\
replace(t, n, s) & = & s, \text{ if } n = \varepsilon \\
 & = & \sigma(t_1, \ldots, t_{i-1}, replace(t_i, n', s), t_{i+1}, \ldots, t_k), \text{ if } n = in' \\
 & & \text{with } i \in I\!N,\ n' \in I\!N^*.
\end{array}
$$

□

The notation of nodes is extended as follows: The root node ε of t is denoted by $root(t)$, too. For every $x \in node(t)$, x is denoted by $x.0$, too. If $x = vi$ with $i \in I\!N_+$, the node v is also denoted by $x.$-1.

Definition 2.3 (Typed, computable functions). Let Ω be a K-sorted set of values. The *set of (typed, computable) functions over* Ω, denoted by $Fun(\Omega)$, is the set $\bigcup_{\tau \in (\mathrm{K}^* \times \mathrm{K})} Fun^\tau(\Omega)$, where for every $\tau = (\kappa_1 \ldots \kappa_k, \kappa_0) \in \mathrm{K}^* \times \mathrm{K}$ with $k \geq 0$, $Fun^\tau(\Omega) = \{f \mid f \text{ is a computable function of the type } f : \Omega^{\kappa_1} \times \ldots \times \Omega^{\kappa_k} \to \Omega^{\kappa_0}\}$.
□

Definition 2.4 (Interpretation of symbols and trees). Let Ω be a K-sorted set of values. An *interpretation of the symbols of* Σ is a family $(\varphi^\tau \mid \tau \in \mathrm{K}^* \times \mathrm{K})$ of computable functions $\varphi^\tau : \Sigma^\tau \to Fun^\tau(\Omega)$.

The *extension of the interpretation of symbols of* Σ *to trees over* Σ *indexed by* Y is a family $(\hat{\varphi}^\tau \mid \tau \in \mathrm{K}^* \times \mathrm{K})$ of mappings $\hat{\varphi}^{(\kappa_1 \ldots \kappa_k, \kappa_0)} : T_\Sigma(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\kappa_0} \to Fun^{(\kappa_1 \ldots \kappa_k, \kappa_0)}(\Omega)$ with $k \geq 0$ and $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$, which are defined inductively on the structure of their arguments.

Let $\tau = (\kappa_1 \ldots \kappa_k, \kappa_0) \in \mathrm{K}^* \times \mathrm{K}$ with $k \geq 0$. For $z \in T_\Sigma(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\kappa_0}$ and $\vec{c} = (c_1, \ldots, c_k) \in \Omega^{\kappa_1} \times \ldots \times \Omega^{\kappa_k}$ define

$$\begin{aligned} \hat{\varphi}^\tau(z)(\vec{c}) &= c_i, \text{ if } z = y_i^{\kappa_i} \text{ for some } i \in \{1, \ldots, k\}. \\ &= \varphi^{(\nu_1 \ldots \nu_l, \kappa_0)}(f)(\hat{\varphi}^{(\kappa_1 \ldots \kappa_k, \nu_1)}(z_1)(\vec{c}), \ldots, \hat{\varphi}^{(\kappa_1 \ldots \kappa_k, \nu_l)}(z_l)(\vec{c})), \\ &\quad \text{if } z = f(z_1, \ldots, z_l) \text{ for some } f \in \Sigma^{(\nu_1 \ldots \nu_l, \kappa_0)} \text{ with } l \geq 0 \\ &\quad \text{and } z_i \in T_\Sigma(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\nu_i}. \end{aligned}$$

□

2.3 Graphs

Definition 2.5 (Directed graph). A *directed graph* is a pair (V, E) consisting of a finite *set* V *of vertices*, and a binary relation $E \subseteq V \times V$ which is called the *set of edges*. □

The following terminology will be used for directed graphs throughout our paper. For an edge $(v_1, v_2) \in E$ the vertex $v_1 \in V$ is called a *direct predecessor of* v_2 and the vertex $v_2 \in V$ is called a *direct successor of* v_1. For a pair $(v_1, v_2) \in E^+ \setminus E$, the vertex $v_1 \in V$ is called an *indirect predecessor of* v_2 and the vertex $v_2 \in V$ is called an *indirect successor of* v_1. If the direction of an edge $(v_1, v_2) \in E$ of an arbitrary directed graph is known from the context, or if the direction of the edge is only of minor interest, then the vertices v_1 and v_2 are called *neighbours*.

Definition 2.6 (Directed acyclic graph). A directed graph $G = (V, E)$ is called *acyclic*, if E^+ does not contain any pair (v, v) with $v \in V$. □

Definition 2.7 (Topological sorting). Let $G = (V, E)$ be a directed acyclic graph. A *topological sorting of* G, denoted by *topsort*(G), is a sequence $v_1 \ldots v_m$ of *all* vertices of G with $m = |V|$, such that for every $i, j \in \{1, \ldots, m\}$, if $(v_i, v_j) \in E$, then $i < j$ holds. □

2.4 Hypergraphs

For the notation of hypergraphs we adapt the basic terminology introduced in [Hab92].

Definition 2.8 (Hypergraph). Let Σ be a set of symbols. A *(directed, edge-labeled) hypergraph over* Σ is a tuple $(V, E, label, source, target, begin, end)$, where V is a finite *set of vertices*, E is a finite set of *hyperedges*, $label : E \to \Sigma$ is the *edge-labeling function*, $source : E \to V^*$ and $target : E \to V^*$ are the *incidence functions*, and $begin \in V^*$ and $end \in V^*$ are words of *external vertices*. The vertices of V, which do not occur in $begin \circ end$, are called *internal vertices*. □

For a given hypergraph h, its components are denoted by V_h, E_h, $label_h$, $source_h$, $target_h$, $begin_h$, and end_h, respectively.

For every hypergraph h, the *interface graph of* h, denoted by $interface(h)$, is the hypergraph $(\{v \mid v \text{ occurs in } begin_h \circ end_h\}, \emptyset, label_h|_{\emptyset}, source_h|_{\emptyset}, target_h|_{\emptyset}, begin_h, end_h)$, i.e., the hypergraph which consists of the external vertices of h only.

A *path in h from vertex v_0 to vertex v_m* is a word $v_0 e_1 v_1 \dots e_m v_m$ in $V_h \circ (E_h \circ V_h)^*$, with $m \geq 0$, such that for every $i \in \{1, \dots, m\}$ the vertex v_{i-1} occurs in $target(e_i)$ and the vertex v_i occurs in $source(e_i)$. A hypergraph h is called *acyclic*, if for every vertex v_0 and for every $m \geq 1$ there is no path $v_0 e_1 v_1 \dots e_m v_m$ in h with $v_m = v_0$.

Let h and g be hypergraphs. The hypergraph g is called a *subgraph of* h, if $V_g \subseteq V_h$, $E_g \subseteq E_h$, and $label_g = label_h|_{E_g}$, $source_g = source_h|_{E_g}$, and $target_g = target_h|_{E_g}$.

If Σ and Γ are disjoint sets of symbols and h is a hypergraph over $\Sigma \cup \Gamma$, then $label_h$ induces a partition of E_h into the sets $E_{h,\Sigma} = \{e \mid e \in E_h \text{ with } label(e) \in \Sigma\}$ of *Σ-labeled edges* and $E_{h,\Gamma} = \{e \mid e \in E_h \text{ with } label(e) \in \Gamma\}$ of *Γ-labeled edges*.

Definition 2.9 (Identification of vertices). Let Σ be a set of symbols. Let h be a hypergraph over Σ. Let $I \subseteq V_h \times V_h$ be a binary relation over the set of vertices of h. Let $\equiv_I$ be the smallest equivalence relation over V_h containing I. Let $class : V_h \to V_h/{\equiv_I}$ be the canonical mapping to the set $V_h/{\equiv_I}$ of equivalence classes. The *hypergraph with vertices identified by I*, denoted by h/I, is the hypergraph $(V_h/{\equiv_I}, E_h, label_h, source', target', begin', end')$ with the following components. For every $e \in E_h$ the incidence functions of h/I are $source'(e)(i) = class(source_h(e)(i))$ with $i \in \{1, \dots, |\, source_h(e)\,|\}$ and $target'(e)(j) = class(target_h(e)(j))$ with $j \in \{1, \dots, |\, target_h(e)\,|\}$. The external vertices of h/I are $begin'(i) = class(begin_h(i))$ with $i \in \{1, \dots, |\, begin_h\,|\}$ and $end'(j) = class(end_h(j))$ with $j \in \{1, \dots, |\, end_h\,|\}$. □

Definition 2.10 (Hyperedge substitution). Let Σ be a set of symbols. Let h be a hypergraph over Σ. Let $e \in E_h$ be an edge of h, and let g be a hypergraph over Σ, such that $|\, begin_g\,| = |\, source_h(e)\,|$ and $|\, end_g\,| = |\, target_h(e)\,|$. The *substitution of e in h by g* is the hypergraph $f/I_{e,g}$ where f has the following components. The set V_f of vertices is $V_f = V_h \cup V_g$. The set E_f of edges is $E_f = (E_h \setminus \{e\}) \cup E_g$. The labeling function is $label_f = (label_h \cup label_g)|_{E_f}$. The incidence functions are $source_f = (source_h \cup source_g)|_{E_f}$ and $target_f = (target_h \cup target_g)|_{E_f}$. The external vertices are $begin_f = begin_h$ and $end_f = end_h$. The identification relation $I_{e,g} \subseteq V_f \times V_f$ induced by the substitution is $I_{e,g} = \{(source_h(e)(i), begin_g(i)) \mid i \in \{1, \dots, |\, source_h(e)\,|\}\} \cup \{(target_h(e)(i), end_g(i)) \mid i \in \{1, \dots, |\, target_h(e)\,|\}\}$. □

2.5 Context-free grammars

Definition 2.11 (Context-free grammar). A *context-free grammar* is a tuple $G_0 = (N, \Sigma, Z, P)$ consisting of a finite *set N of nonterminal symbols*, a finite *set Σ of terminal symbols*, where $N \cap \Sigma = \emptyset$, a designated *start symbol* $Z \in N$, and a finite *set $P \subseteq N \times (N \cup \Sigma)^*$ of productions* of the form $X_0 \to w_0 X_1 w_1 \dots X_n w_n$ with $n \geq 0$, $X_0, \dots, X_n \in N$, and $w_0, \dots, w_n \in \Sigma^*$. □

In the following let $G_0 = (N, \Sigma, Z, P)$ be a context-free grammar.

Definition 2.12 (Derivation relation). The *derivation relation of* G_0,denoted by $\Rightarrow_{G_0}$, is the smallest set $\bigcup_{p \in P} \Rightarrow_p$, such that for every $p \in P$ of the form $X_0 \to w_0 X_1 w_1 \dots X_n w_n$ and for every $u, v \in (N \cup \Sigma)^*$, $(u X_0 v, u w_0 X_1 w_1 \dots X_n w_n v) \in \Rightarrow_p$ holds. □

Definition 2.13 (X-derivation tree). The derivation trees of G_0 are inductively defined as follows. If $X_0 \to w_0X_1w_1 \dots X_nw_n$ with $n \geq 0$ is a production of G_0, where for every $i \in \{0, \dots, n\}$ the $w_i \in \Sigma^*$ has the form $w_i = a_{i,1} \dots a_{i,r_i}$ with $r_i = |\, w_i \,|$ and $a_{i,1}, \dots, a_{i,r_i} \in \Sigma$, and if for every $i \in \{1, \dots, n\}$ the tree t_i is an X_i-derivation tree of G_0, then the tree $X_0(a_{0,1}, \dots, a_{0,r_0}, t_1, a_{1,1}, \dots, a_{1,r_1}, \dots, t_n, a_{n,1}, \dots, a_{n,r_n})$ is an X_0-*derivation tree of* G_0. The *set of all derivation trees of* G_0, denoted by $DT(G_0)$, is the set of all Z-derivation trees of G_0. □

To describe properties of derivation trees, the notation already introduced for sorted trees is adapted: For every $t \in DT(G_0)$, $node(t)$ denotes the *set of nodes of* t, $root(t)$ denotes the *root node*, and for every $x \in node(t)$, $label_t(x)$ denotes the *label of* x. Additionally, for every $x \in node(t)$ the *production applied at* x is denoted by $prod_t(x)$.

Definition 2.14 (Language generated by a grammar). The *language of* G_0,denoted by $L(G_0)$, is the set $\{w \mid w \in \Sigma^*, Z {\Rightarrow_{G_0}}^* w\}$. □

3 Macro Attribute Grammars

We define mags and the translation relation induced by them. The concept of translation is based on associating attributes with the nonterminals of a context-free grammar and on assigning values to the instances of attributes at the occurrences of nonterminals in derivation trees, i.e., *decorating* the derivation trees. The existence of a unique decoration of a derivation tree is assured, if we restrict the mag to be noncircular. We define the class of ordered mags, which is a subclass of noncircular mags investigated throughout the rest of this paper.

To illustrate the formal concepts, we give an example mag and introduce a graphical notation to visualize dependency relations specified by a given mag.

3.1 Basics of Macro Attribute Grammars

Mags are a generalization of ags. The generalization is the result of the introduction of attributes whose values can be functions. To study the properties of a syntax-directed translation model in which function values are allowed, macro attributed tree transducers have been introduced in [KV94].

Definition 3.1 (Mag). An *mag* is a tuple $M = (G_0, D, B, R)$ consisting of:

- a *context-free grammar* G_0,
- a *semantic domain* D,
- an *attribute description* B, and
- a *description* R *of semantic rules*.

G_0 is a tuple (N, Σ, Z, P) according to Definition 2.11.
D is a tuple $(\mathrm{K}, \Omega, \Phi, \varphi)$ consisting of:

- a finite set K of *sorts*,
- a K-sorted set Ω of *values*,
- a finite $(\mathrm{K}^* \times \mathrm{K})$-sorted set Φ of *function symbols*, and
- an *interpretation* φ of the symbols in Φ due to Definition 2.4.

B is a tuple $(S\text{-}Att, I\text{-}Att, A, \alpha_0)$ consisting of:

- two disjoint finite $(\mathrm{K}^* \times \mathrm{K})$-sorted sets $S\text{-}Att$ and $I\text{-}Att$, where the elements of $S\text{-}Att$ are called *synthesized attributes*, and the elements of $I\text{-}Att$ are called *inherited attributes*,
- a mapping $A : N \to \mathcal{P}(S\text{-}Att \cup I\text{-}Att)$, which maps every nonterminal symbol $X \in N$ to a set of attributes, where $A(Z) \cap I\text{-}Att = \emptyset$, and
- a designated attribute $\alpha_0 \in A(Z)$ with $sort(\alpha_0) \in (\{\varepsilon\} \times \mathrm{K})$, which is called the *meaning attribute*.

Let $X_0 \to w_0 X_1 w_1 \dots X_n w_n$ be a production of G_0 (cf. Definition 2.11). The $(\mathrm{K}^* \times \mathrm{K})$-sorted set $A(p)$ of *attribute occurrences of* p is the set

$$A(p) = \{\langle a, j\rangle \mid 0 \le j \le n, a \in A(X_j)\},$$

where $sort(\langle a, j\rangle) = sort(a)$ for every $\langle a, j\rangle \in A(p)$.
The set $A(p)$ is partitioned into two disjoint subsets:

- the set $inner(p)$ of *inner attribute occurrences of* p:

$$inner(p) = \{\langle a, 0\rangle \mid a \in A(X_0) \cap S\text{-}Att\} \cup \{\langle a, j\rangle \mid a \in A(X_j) \cap I\text{-}Att, 1 \le j \le n\}$$

- and the set $outer(p)$ of *outer attribute occurrences of* p:

$$outer(p) = \{\langle a, 0\rangle \mid a \in A(X_0) \cap I\text{-}Att\} \cup \{\langle a, j\rangle \mid a \in A(X_j) \cap S\text{-}Att, 1 \le j \le n\}.$$

R is a pair (Y, RP) consisting of:

- a finite K-sorted set Y of parameters and
- a family $RP = (\ R(p) \mid p \in P\ )$ of semantic rules.

  For every production p the set $R(p)$ of semantic rules contains for every inner attribute occurrence $\langle a, j\rangle \in inner(p)$ with $sort(\langle a, j\rangle) = (\kappa_1 \dots \kappa_k, \kappa_0)$, $k \ge 0$, $\kappa_0, \dots, \kappa_k \in K$ exactly one semantic rule of the form

$$\langle a, j\rangle(y_1^{\kappa_1}, \dots, y_k^{\kappa_k}) = \zeta_{\langle a,j\rangle},$$

  where $\zeta_{\langle a,j\rangle} \in T_{\Phi \cup outer(p)}(\{y_1^{\kappa_1}, \dots, y_k^{\kappa_k}\})^{\kappa_0}$. □

The concept of inner and outer attribute occurrences of a production, which is the base for the definition of the semantic rules of the production, is illustrated by Figure 2, which shows the production $p = L \to LL$, and its attribute occurrences derived from the sets $A(L) \cap S\text{-}Att = \{l, v\}$ and $A(L) \cap I\text{-}Att = \{r\}$.

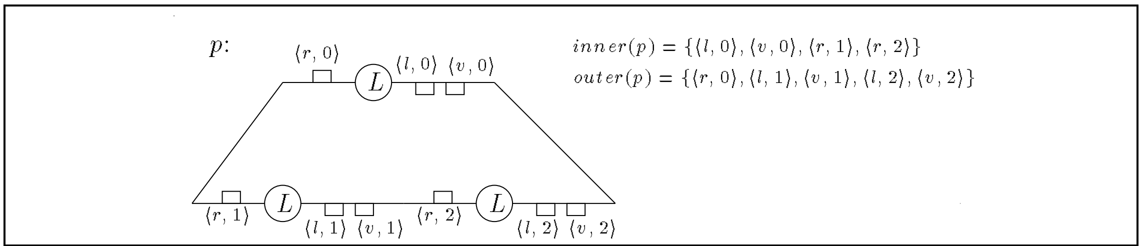


Figure 2: Illustration of inner and outer attribute occurrences.

The set of semantic rules of a production p induces a dependency relation over the attribute occurrences of p. Intuitively, given a semantic rule $r = (\langle a, j\rangle(y_1, \dots, y_k) = \zeta_{\langle a,j\rangle})$, the attribute occurrence $\langle a, j\rangle \in A(p)$ on the left hand side of r depends on every attribute occurrence $\langle b, i\rangle \in A(p)$ which is used in the right hand side $\zeta_{\langle a,j\rangle}$ of r, i.e., it depends on every $\langle b, i\rangle \in sym(\zeta_{\langle a,j\rangle}) \cap A(p)$. But note that the rule r does not induce any dependencies amongst the attribute occurrences, which are nested in $\zeta_{\langle a,j\rangle}$.

As usual, the existing dependencies between the attribute occurrences of a production p are represented as the dependency graph of p. In the following, let $M = (G_0, D, B, R)$ be an mag with $G_0 = (N, \Sigma, Z, P)$ as underlying context-free grammar and with K as set of sorts of the semantic domain.

Definition 3.2 (Dependency graph of a production). Let $p \in P$ be a production of G_0. The *dependency graph of* p, denoted by $D(p)$, is the directed graph $D(p) = (A(p), E(p))$, where $E(p) = \{(\langle b, i\rangle, \langle a, j\rangle) \mid (\langle a, j\rangle(y_1, \ldots, y_k) = \zeta_{\langle a,j\rangle}) \in R(p),\ \langle b, i\rangle \in (sym(\zeta_{\langle a,j\rangle}) \cap A(p))\}$. □

3.2 An example macro attribute grammar

To illustrate the introduced concepts, the following mag M_{10} is used throughout our paper. M_{10} transforms every binary number in $\{0,1\}^* \setminus \{\varepsilon\}$ into the corresponding number (which we will represent in its decimal notation).

Example 3.3. $M_{10} = (G_0, D, B, R)$, where

- $G_0 = (N, \Sigma, Z, P)$, where

$$\begin{aligned} &N = \{Z, L\},\\ &\Sigma = \{0, 1\},\ \text{and}\\ &P = \{p_1, p_2, p_3, p_4\},\ \text{where}\\ &\quad p_1 = Z \to L, \quad p_2 = L \to LL, \quad p_3 = L \to 0,\ \text{and} \quad p_4 = L \to 1. \end{aligned}$$

- $D = (\mathrm{K}, \Omega, \Phi, \varphi)$, where

$$\begin{aligned} &\mathrm{K} = \{nat\},\\ &\Omega = \Omega^{nat} = I\!N,\\ &\Phi = \{null^{(\varepsilon,nat)}, s^{(nat,nat)}\},\ \text{and}\\ &\varphi = (\varphi^{(\varepsilon,nat)}, \varphi^{(nat,nat)})\ \text{with}\\ &\quad \varphi^{(\varepsilon,nat)}(null)() = 0,\ \text{and for every } e \in I\!N, \varphi^{(nat,nat)}(s)(e) = e + 1 \text{ holds.} \end{aligned}$$

- $B = (S\text{-}Att, I\text{-}Att, A, \alpha_0)$, where

$$\begin{aligned} &S\text{-}Att = \{\alpha_0^{(\varepsilon,nat)}, l^{(nat,nat)}, v^{(nat,nat)}\},\\ &I\text{-}Att = \{r^{(nat,nat)}\},\ \text{and}\\ &A(Z) = \{\alpha_0\},\ A(L) = \{l, v, r\}. \end{aligned}$$

In Figure 3 the naming conventions of the attributes and the intended meaning of the attributes are illustrated. Consider the attributes associated with the root node of an L-derivation tree and the word $w_L = b_j \ldots b_i$, which is the front of this L-derivation tree. Intuitively, the instance of the inherited attribute r contains the weight of the binary digit b_i on the right side of w_L, i.e., r holds the value 2^i. The instance of the synthesized

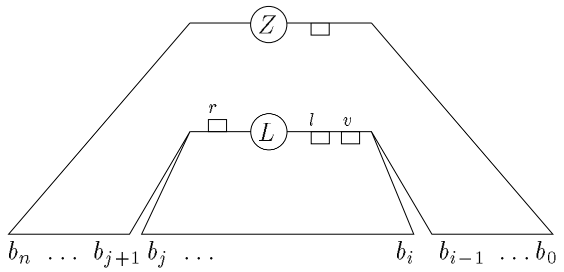


Figure 3: Associated meaning of instances of the attributes r, l, and v.

attribute l contains the weight of the digit b_j on the left side of w_L, i.e., l holds the value 2^j. This value has to be computed in order to compute the weight of the digit b_{j+1}. Clearly, the instance of the synthesized attribute v holds the value of the interpretation of the whole word w_L with respect to the weight r.

- $R = (Y, RP)$, where

$$
\begin{array}{rlrcl}
Y = \{y^{nat}\}, \text{ and} & & & & \\
RP = (R(p) \mid p \in P) \text{ with} & R(p_1) = & \{ \ \langle r,1\rangle(y^{nat}) & = & s(y^{nat}), \\
& & \langle \alpha_0, 0\rangle & = & \langle v,1\rangle(null) \ \} \\
& R(p_2) = & \{ \ \langle r,1\rangle(y^{nat}) & = & \langle l,2\rangle(\langle l,2\rangle(y^{nat})), \\
& & \langle r,2\rangle(y^{nat}) & = & \langle r,0\rangle(y^{nat}), \\
& & \langle l,0\rangle(y^{nat}) & = & \langle l,1\rangle(y^{nat}), \\
& & \langle v,0\rangle(y^{nat}) & = & \langle v,1\rangle(\langle v,2\rangle(y^{nat})) \ \} \\
& R(p_3) = & \{ \ \langle l,0\rangle(y^{nat}) & = & \langle r,0\rangle(y^{nat}), \\
& & \langle v,0\rangle(y^{nat}) & = & y^{nat} \ \} \\
& R(p_4) = & \{ \ \langle l,0\rangle(y^{nat}) & = & \langle r,0\rangle(y^{nat}), \\
& & \langle v,0\rangle(y^{nat}) & = & \langle r,0\rangle(y^{nat}) \ \}
\end{array}
$$

Intuitively, the computation of the number of a given word of binary digits is realized by the composition of s-functions, where s is a mnemonic notation for "successor". With this idea in mind, the semantic rules of the productions can be understood as follows.

The first rule of p_1 initializes the computation of the weight of every digit by assigning the s-function to $\langle r,1\rangle$ in correspondence to the weight of b_0 (compare Figure 3). The second rule of p_1 is the "output rule" and terminates the composition of s-functions by assigning the computed number to α_0 at the root.

The rules of p_2 realize the compositions. The first rule determines the weight $\langle r,1\rangle$ of the right digit in the left subtree as the double weight $\langle l,2\rangle$ of the left digit in the right subtree. The decimal value $\langle v,0\rangle$ is the sum of the values of the left and the right subtree, i.e., the composition of the accumulated functions $\langle v,1\rangle$ and $\langle v,2\rangle$.

The rules of the productions p_3 and p_4 have to determine the values of a single digit only. In the case of p_4 this is just the weight of the digit, which is already computed in $\langle r,0\rangle$. In the case of p_3 the digit has value 0, i.e., instead of the application of s-functions the identity function is generated for $\langle v,0\rangle$.

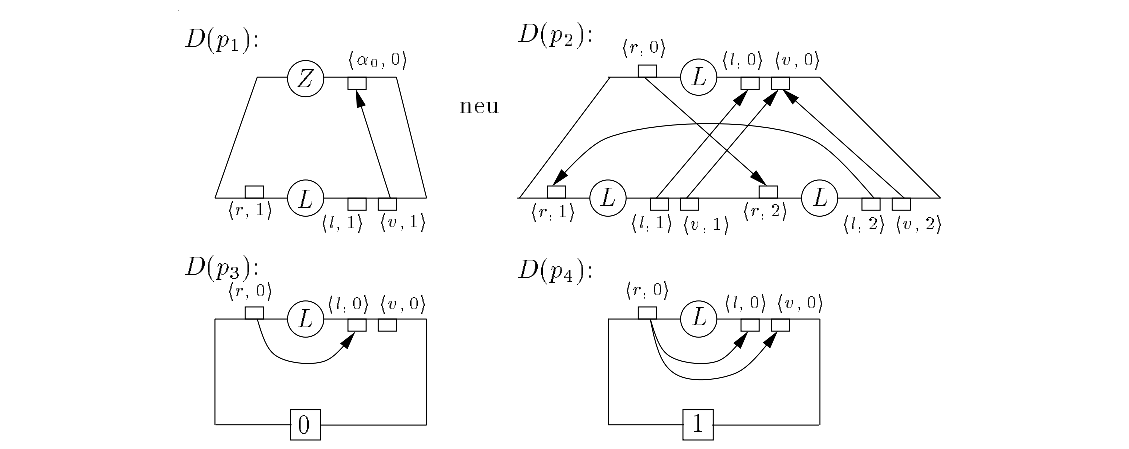


Figure 4: Dependency graphs of the productions p_1, p_2, p_3, and p_4.

The dependency graphs of the productions p_1, p_2, p_3, and p_4 are illustrated in Figure 4. In fact, the diagrams of Figure 4 show more than the pure dependency graphs of the productions. Nevertheless, we refer to such diagrams as dependency graphs and denote them by $D(p)$ for every $p \in P$, because they are the illustrations of productions and attribute occurrences of Figure 2 *extended* by arrows for the directed edges of the dependency graphs. ■

3.3 The translation of derivation trees

The translation of a derivation tree is based on assigning functions over values of the semantic domain to instances of attributes in the derivation tree. Every node of a derivation tree is labeled by a nonterminal symbol. Since every nonterminal symbol of an mag owns a set of attributes, the attributes of a nonterminal symbol X are instantiated with every occurrence of X, too.

Let $t \in DT(G_0)$ and let $x \in node(t)$ with $label_t(x) \in N$ be a node of t.

Definition 3.4 (Attribute instance). The *set of attribute instances of* x, denoted by $A(x)$, is the $(\mathrm{K}^* \times \mathrm{K})$-sorted set $A(x) = \{\langle a, x\rangle \mid a \in A(label_t(x))\}$, where for every $\langle a, x\rangle \in A(x)$, $sort(\langle a, x\rangle) = sort(a)$. The *set of attribute instances of* t, denoted by $A(t)$, is the $(\mathrm{K}^* \times \mathrm{K})$-sorted set $A(t) = \bigcup_{x \in node(t), label_t(x) \in N} A(x)$. □

Similar to the instantiation of the attributes, the terms on the right hand sides of the semantic rules of a production are instantiated by associating them with a node of the derivation tree.

Definition 3.5 (Embedding). Let $p = prod_t(x)$ be the production applied at x. Let $\zeta \in T_{\Phi \cup outer(p)}(Y)^{\kappa_0}$ with $\kappa_0 \in \mathrm{K}$. The *embedding of* ζ *at* x, denoted by $[\zeta]_x$, is the term in $T_{\Phi \cup A(t)}(Y)^{\kappa_0}$ defined as follows:

$$[\zeta]_x = \zeta, \text{ if } \zeta \in Y.$$
$$= f([\zeta_1]_x, \ldots, [\zeta_k]_x), \text{ if } \zeta = f(\zeta_1, \ldots, \zeta_k) \text{ and } f \in \Phi^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$$
$$\text{with } k \geq 0 \text{ and for every } i \in \{1, \ldots, k\},\ \kappa_i \in \mathrm{K} \text{ and } \zeta_i \in T_{\Phi \cup outer(p)}(Y)^{\kappa_i} \text{ holds.}$$

$= \langle b, x.j\rangle([\zeta_1]_x, \ldots, [\zeta_k]_x)$, if $\zeta = \langle b, j\rangle(\zeta_1, \ldots, \zeta_k)$ and $\langle b, j\rangle \in outer(p)^{(\kappa_1\ldots\kappa_k,\kappa_0)}$ with $k \geq 0$ and for every $i \in \{1, \ldots, k\}$, $\kappa_i \in \mathrm{K}$ and $\zeta_i \in T_{\Phi\cup outer(p)}(Y)^{\kappa_i}$ holds. □

The instances of the right hand sides of the semantic rules are used as the right hand sides of the instances of semantic rules. The instances of semantic rules are called *semantic equations.*

Definition 3.6 (Semantic equation). Let $p = prod_t(x)$. Let $\langle a, i\rangle(y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}) = \zeta_{\langle a,i\rangle}$ be a semantic rule of p. Then the equation $\langle a, x.i\rangle(y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}) = [\zeta_{\langle a,i\rangle}]_x$ is a *semantic equation of* t (Recall that $x.0 = x$ for every node x.). The set of all semantic equations of t is denoted by $R(t)$. □

The same way the set $R(p)$ of semantic rules of a production p induces a dependency relation over attribute occurrences, the set $R(t)$ of semantic equations induces a dependency relation over the attribute instances of t, which is represented as the dependency graph of t.

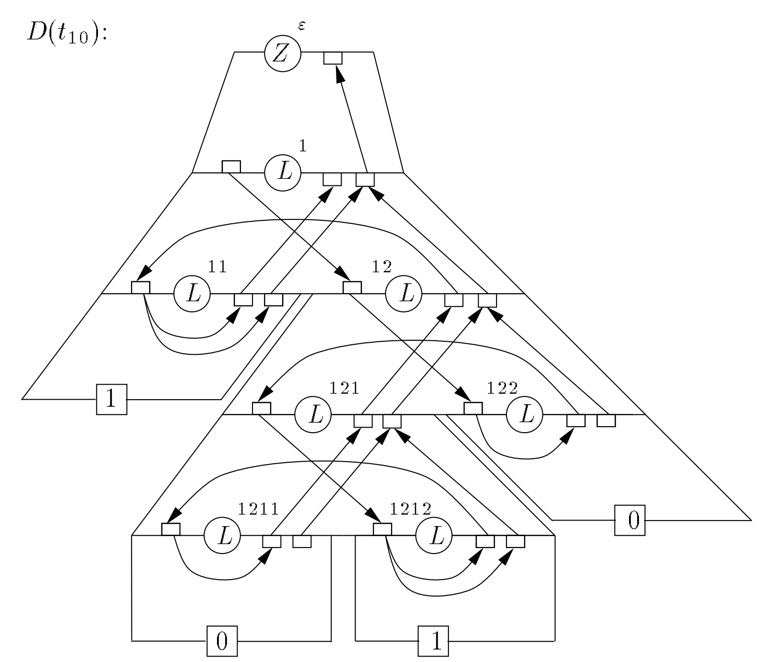


Figure 5: Dependency graph of the derivation tree t_{10} of the word 1010.

Definition 3.7 (Dependency graph of a derivation tree). The *dependency graph of* t, denoted by $D(t)$, is the directed graph $D(t) = (A(t), E(t))$, where
$E(t) = \{(\langle b, x.j\rangle, \langle a, x.i\rangle) \mid x \in node(t), (\langle a, x.i\rangle(y_1, \ldots, y_k) = [\zeta_{\langle a,i\rangle}]_x) \in R(t), \langle b, x.j\rangle \in (sym([\zeta_{\langle a,i\rangle}]_x) \cap A(t))\ \}$ □

Example 3.8 (continued). A word of the language of the example grammar M_{10} specified in Section 3.2 is the word $w = 1010$. A derivation tree for w is the tree t_{10} shown in Figure 5. The dependency graph $D(t_{10})$ is derived from the following set

$R(t_{10})$ of the semantic equations of t_{10}, where the parameter y^{nat} is abbreviated by y.

$$\begin{aligned}
\langle r,1\rangle(y) &= s(y), & \langle v,1211\rangle(y) &= y,\\
\langle r,12\rangle(y) &= \langle r,1\rangle(y), & \langle l,121\rangle(y) &= \langle l,1211\rangle(y),\\
\langle r,122\rangle(y) &= \langle r,12\rangle(y), & \langle v,121\rangle(y) &= \langle v,1211\rangle(\langle v,1212\rangle(y)),\\
\langle l,122\rangle(y) &= \langle r,122\rangle(y), & \langle l,12\rangle(y) &= \langle l,121\rangle(y),\\
\langle v,122\rangle(y) &= y, & \langle v,12\rangle(y) &= \langle v,121\rangle(\langle v,122\rangle(y)),\\
\langle r,121\rangle(y) &= \langle l,122\rangle(\langle l,122\rangle(y)), & \langle r,11\rangle(y) &= \langle l,12\rangle(\langle l,12\rangle(y)),\\
\langle r,1212\rangle(y) &= \langle r,121\rangle(y), & \langle l,11\rangle(y) &= \langle r,11\rangle(y),\\
\langle l,1212\rangle(y) &= \langle r,1212\rangle(y), & \langle v,11\rangle(y) &= \langle r,11\rangle(y),\\
\langle v,1212\rangle(y) &= \langle r,1212\rangle(y), & \langle l,1\rangle(y) &= \langle l,11\rangle(y),\\
\langle r,1211\rangle(y) &= \langle l,1212\rangle(\langle l,1212\rangle(y)), & \langle v,1\rangle(y) &= \langle v,11\rangle(\langle v,12\rangle(y)),\\
\langle l,1211\rangle(y) &= \langle r,1211\rangle(y), & \langle \alpha_0,\varepsilon\rangle &= \langle v,1\rangle(null).
\end{aligned}$$

■

Assigning to every attribute instance of a derivation tree a value yields a *decoration* of the derivation tree, where the sort of an attribute instance restricts the values which can be assigned. Only those values are allowed which have the proper type.

Definition 3.9 (Decoration). A *decoration* val_t *of* t is a family $(val_t^\tau \mid \tau \in \mathrm{K}^* \times \mathrm{K})$ of mappings $val_t^\tau : A(t)^\tau \to Fun^\tau(\Omega)$. □

Example 3.10 (continued). The following family of mappings is a decoration of the derivation tree t_{10} in Figure 5: $val_{t_{10}} = (val_{t_{10}}^{(\varepsilon,nat)}, val_{t_{10}}^{(nat,nat)})$, where the mappings $val_{t_{10}}^{(\varepsilon,nat)}$ and $val_{t_{10}}^{(nat,nat)}$ are defined as follows. For every $n \in I\!N$ it holds, that:

$$\begin{aligned}
val_{t_{10}}^{(nat,nat)}(\langle r,1\rangle)(n) &= n+1, & val_{t_{10}}^{(nat,nat)}(\langle v,1211\rangle)(n) &= n,\\
val_{t_{10}}^{(nat,nat)}(\langle r,12\rangle)(n) &= n+1, & val_{t_{10}}^{(nat,nat)}(\langle l,121\rangle)(n) &= n+4,\\
val_{t_{10}}^{(nat,nat)}(\langle r,122\rangle)(n) &= n+1, & val_{t_{10}}^{(nat,nat)}(\langle v,121\rangle)(n) &= n+2,\\
val_{t_{10}}^{(nat,nat)}(\langle l,122\rangle)(n) &= n+1, & val_{t_{10}}^{(nat,nat)}(\langle l,12\rangle)(n) &= n+4,\\
val_{t_{10}}^{(nat,nat)}(\langle v,122\rangle)(n) &= n, & val_{t_{10}}^{(nat,nat)}(\langle v,12\rangle)(n) &= n+2,\\
val_{t_{10}}^{(nat,nat)}(\langle r,121\rangle)(n) &= n+2, & val_{t_{10}}^{(nat,nat)}(\langle r,11\rangle)(n) &= n+8,\\
val_{t_{10}}^{(nat,nat)}(\langle r,1212\rangle)(n) &= n+2, & val_{t_{10}}^{(nat,nat)}(\langle l,11\rangle)(n) &= n+8,\\
val_{t_{10}}^{(nat,nat)}(\langle l,1212\rangle)(n) &= n+2, & val_{t_{10}}^{(nat,nat)}(\langle v,11\rangle)(n) &= n+8,\\
val_{t_{10}}^{(nat,nat)}(\langle v,1212\rangle)(n) &= n+2, & val_{t_{10}}^{(nat,nat)}(\langle l,1\rangle)(n) &= n+8,\\
val_{t_{10}}^{(nat,nat)}(\langle r,1211\rangle)(n) &= n+4, & val_{t_{10}}^{(nat,nat)}(\langle v,1\rangle)(n) &= n+10,\\
val_{t_{10}}^{(nat,nat)}(\langle l,1211\rangle)(n) &= n+4, \quad \text{and} & val_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0,\varepsilon\rangle) &= 10.
\end{aligned}$$

■

For the specification of the translation of every derivation tree, we need decorations which do not only regard the sorts of the attribute instances, but which are solutions of the semantic equations, too. We call these decorations *admissible*.

Definition 3.11 (Admissible decoration). A decoration val_t is called *admissible*, if for every semantic equation $\langle a, x.i\rangle(y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}) = [\zeta_{\langle a,i\rangle}]_x$ with $sort(\langle a, x.i\rangle) = (\kappa_1 \ldots \kappa_k, \kappa_0)$, $k \geq 0$, and $\kappa_0, \kappa_1, \ldots, \kappa_k \in \mathrm{K}$ the equality

$$val_t^{(\kappa_1 \ldots \kappa_k, \kappa_0)}(\langle a, x.i\rangle) = \Theta^{(\kappa_1 \ldots \kappa_k, \kappa_0)}([\zeta_{\langle a,i\rangle}]_x)$$

holds, i.e., the two functions are equal. The mapping $\Theta^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$ is a member of the family $(\Theta^\tau \mid \tau \in \mathrm{K}^* \times \mathrm{K})$ of mappings $\Theta^{(\kappa_1 \ldots \kappa_k, \kappa_0)} : T_{\Phi \cup A(t)}(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\kappa_0} \to Fun^{(\kappa_1 \ldots \kappa_k, \kappa_0)}(\Omega)$, where the members of this family are defined simultaneously on the structure of their argument. Let $\tau = (\kappa_1 \ldots \kappa_k, \kappa_0) \in \mathrm{K}^* \times \mathrm{K}$ with $k \geq 0$ and $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$. For $z \in T_{\Phi \cup A(t)}(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\kappa_0}$ and $\vec{c} = (c_1, \ldots, c_k) \in \Omega^{\kappa_1} \times \ldots \times \Omega^{\kappa_k}$ define

$$\begin{aligned}
\Theta^\tau(z)(\vec{c}) &= c_i, \text{ if } z = y_i^{\kappa_i} \text{ for some } i \in \{1, \ldots, k\}.\\
&= \varphi^{(\nu_1 \ldots \nu_l, \kappa_0)}(f)(\Theta^{(\kappa_1 \ldots \kappa_k, \nu_1)}(z_1)(\vec{c}), \ldots, \Theta^{(\kappa_1 \ldots \kappa_k, \nu_l)}(z_l)(\vec{c})),\\
&\quad \text{if } z = f(z_1, \ldots, z_l) \text{ for some } f \in \Phi^{(\nu_1 \ldots \nu_l, \kappa_0)} \text{ with } l \geq 0, \nu_1, \ldots, \nu_l \in \mathrm{K},\\
&\quad \text{and } z_i \in T_{\Phi \cup A(t)}(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\nu_i} \text{ for every } i \in \{1, \ldots, l\}.\\
&= val_t^{(\nu_1 \ldots \nu_l, \kappa_0)}(\langle b, x.j\rangle)(\Theta^{(\kappa_1 \ldots \kappa_k, \nu_1)}(z_1)(\vec{c}), \ldots, \Theta^{(\kappa_1 \ldots \kappa_k, \nu_l)}(z_l)(\vec{c}))\\
&\quad \text{if } z = \langle b, x.j\rangle(z_1, \ldots, z_l) \text{ for some } \langle b, x.j\rangle \in A(t)^{(\nu_1 \ldots \nu_l, \kappa_0)} \text{ with } l \geq 0,\\
&\quad \nu_1, \ldots, \nu_l \in \mathrm{K}, \text{ and } z_i \in T_{\Phi \cup A(t)}(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\nu_i} \text{ for every } i \in \{1, \ldots, l\}. \square
\end{aligned}$$

Example 3.12 (continued). The decoration $val_{t_{10}}$ given in Example 3.10 is admissible. In order to demonstrate the admissibility of $val_{t_{10}}$, we calculate for the semantic equation $\langle \alpha_0, \varepsilon\rangle = \langle v, 1\rangle(null)$ the value of $\Theta^{(\varepsilon,nat)}(\langle v, 1\rangle(null))$.

$$\begin{aligned}
\Theta^{(\varepsilon,nat)}(\langle v, 1\rangle(null)) &= val_{t_{10}}^{(nat,nat)}(\langle v, 1\rangle)(\Theta^{(\varepsilon,nat)}(null))\\
&= val_{t_{10}}^{(nat,nat)}(\langle v, 1\rangle)(\varphi^{(\varepsilon,nat)}(null))\\
&= val_{t_{10}}^{(nat,nat)}(\langle v, 1\rangle)(0)\\
&= 0 + 10\\
&= 10
\end{aligned}$$

Thus, $val_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0, \varepsilon\rangle) = \Theta^{(\varepsilon,nat)}(\langle v, 1\rangle(null))$. The calculation according to the other semantic equations of t_{10} is similar. ■

Without specifying a method for the computation of admissible decorations for the derivation trees of a given mag M, we define in general an *attribute evaluator for M* to be an algorithm, which computes for every derivation tree $t \in DT(G_0)$ an admissible decoration val_t.

For every derivation tree the existence and the uniqueness of an admissible decoration is assured, if we restrict the mag to be noncircular.

Definition 3.13 (Noncircular mag). An mag M is called *noncircular*, if for every derivation tree $t \in DT(G_0)$ its dependency graph $D(t)$ is acyclic. The *class of noncircular mags* is denoted by $NCMAG$. □

The circularity test for ags can be easily adapted to a circularity test for mags (cf. [KV94]).

Every noncircular mag specifies a mapping between every derivation tree of its underlying context free grammar and its admissible decoration. From the concept of the admissible decoration of a derivation tree we derive the concept of the *translation specified by an mag* by considering for every derivation tree t the value of the instance $\langle \alpha_0, root(t)\rangle$ of the meaning attribute at the root of t.

Definition 3.14 (Tree-to-value translation). For every $M \in NCMAG$ the *tree-to-value translation specified by* M, denoted by $\tau_{tv}(M)$, is the mapping $\tau_{tv}(M) : DT(G_0) \rightarrow \Omega^{sort(\alpha_0)}$ defined by $\tau_{tv}(M)(t) = val_t(\langle \alpha_0, \varepsilon\rangle)$ for every $t \in DT(G_0)$. □

Example 3.15 (continued). The mag M_{10} is noncircular, and hence, the tree-to-value translation exists and it is unique. It follows from $val_{t_{10}}(\langle \alpha_0, \varepsilon\rangle) = 10$, that $\tau_{tv}(M_{10})(t_{10}) = 10$. ■

3.4 The property of orderedness

Intuitively, an mag is called *ordered*, if for every nonterminal symbol of the grammar an order of its attributes exists, such that for every *instance* of the nonterminal symbol in *every* derivation tree of the grammar the corresponding attribute instances can be evaluated according to this order.

Clearly, the predetermined orders of the attribute instances are the reason for the attractiveness of ordered mags. Very efficient evaluators, which apply the orders to determine the sequence in which the attribute instances have to be picked up to decorate a given derivation tree, can be constructed automatically.

The idea of orderedness was firstly discussed by Kastens, who introduced *ordered ags* in [Kas80]. Furthermore, the property of orderedness is studied very extensively by Engelfriet and Filé in [EF82]. In [Jok96] the property of orderedness was adapted to mags.

The main step of testing the orderedness of a given grammar is to overlay all dependencies between the attributes of a nonterminal symbol, which can be found by combining the dependency graphs of the productions properly. To obtain the proper combinations of dependency graphs the dependency graphs, which reflect the dependencies of inner attribute occurrences on outer attribute occurrences, have to be extended by graphs, which reflect for every nonterminal symbol occurrence the dependencies of its outer attribute occurrences on its inner attribute occurrences, i.e., we have to "close the gaps" in every dependency graph.

Definition 3.16 (Extended dependency graph). Let $p = X_0 \rightarrow w_0X_1w_1 \ldots X_nw_n$ be a production of G_0. For every $i \in \{0, \ldots, n\}$ let $D_i = (A(X_i), \rightarrow_i)$ with $\rightarrow_i \subseteq A(X_i) \times A(X_i)$ be a graph over the set of attributes of X_i. Let $D(p) = (A(p), E(p))$ be the dependency graph of p. The *extended dependency graph of* p, denoted by $D(p)[D_0; D_1, \ldots, D_n]$, is the graph $(A(p), E)$ with the edge relation $E = E(p) \cup \{(\langle a, i\rangle, \langle b, i\rangle) \mid (a, b) \in \rightarrow_i, i \in \{0, \ldots, n\}\}$. □

The existing attribute dependencies of every nonterminal symbol are collected in the *is*/*si*-graphs of the nonterminal symbols.

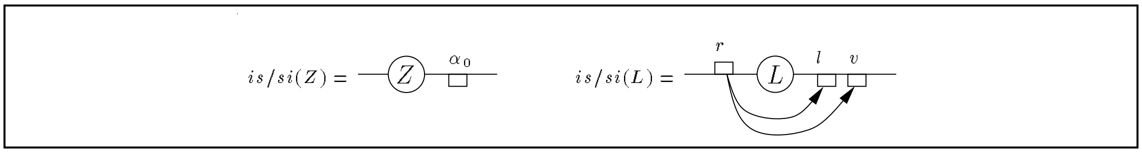


Figure 6: The graphs $is/si(Z)$ and $is/si(L)$.

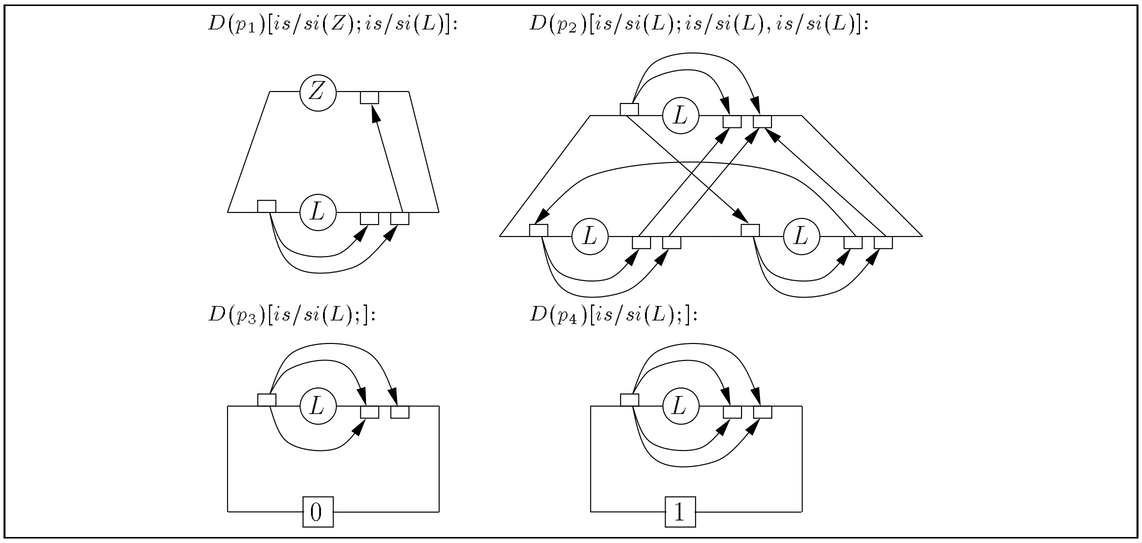


Figure 7: Extended dependency graphs of the productions p_1, p_2, p_3, and p_4.

Definition 3.17 (is/si-graph). Let $N = \{X_1, \ldots, X_m\}$ be the set of nonterminal symbols of G_0. The *is/si–graphs of* $X_1, \ldots, X_m$, denoted by $is/si(X_1), \ldots, is/si(X_m)$, are the smallest graphs $g_1 = (A(X_1), E(X_1)), \ldots, g_m = (A(X_m), E(X_m))$, for which the following holds: for every production $p = X_{i_0} \to w_0 X_{i_1} w_i \ldots X_{i_n} w_n$ with $i_0, \ldots, i_n \in \{1, \ldots, m\}$, and for every $j \in \{0, \ldots, n\}$ it holds that, if $(\langle a, j\rangle, \langle b, j\rangle) \in E^+$, where E is the edge relation of $D(p)[g_{i_0}; g_{i_1}, \ldots, g_{i_n}]$, then $(a, b) \in E(X_{i_j})$ holds. □

Here, we do not work out an algorithm to test the orderedness of a given mag and to compute the is/si-graphs of the nonterminal symbols of the grammar, because the algorithm were literally the same as the algorithm for ordered ags. The algorithm can be found in [Jok96], which deals with mags, and in [KV97], which deals with ags, for instance.

Example 3.18 (continued). In Figure 6 the is/si-graphs of the nonterminal symbols Z and L of the example mag M_{10} are shown. The dependency graphs of the productions p_1, p_2, p_3, and p_4 extended by the is/si-graphs of Z and L are shown in Figure 7, where the *extending* is/si-graphs are illustrated by the additional outer edges. ■

If the is/si-graph of a nonterminal symbol is acyclic, then the is/si-graph induces an order of all attributes of the nonterminal symbol, which was mentioned at the beginning of this section. Such an order is obtained by transforming the partial order, which is given by the edge relation of the is/si-graph, into a total order, which is called *evaluation order*, because it determines the order in which the attributes are evaluated.

Definition 3.19 (Evaluation order of a nonterminal symbol). Let $X \in N$ be a nonterminal symbol of G_0. Let $is/si(X) = (A(X), E(X))$ be the is/si–graph of X. An *evaluation order of X derived from $is/si(X)$* is a total order $\tilde{\beta}_1\tilde{\alpha}_1 \ldots \tilde{\beta}_{k(X)}\tilde{\alpha}_{k(X)}$ of $A(X)$ with minimal $k(X) \in I\!N$, such that for every $j \in \{1, \ldots, k(X)\}$ the following holds: $\tilde{\beta}_j$ is an arbitrary total order of all inherited attributes, which are the maximal elements of the relation $E(X)\lceil A(X) \setminus \{a \mid a \in A(X), a \text{ occurs in } \tilde{\alpha}_j\tilde{\beta}_{j+1}\tilde{\alpha}_{j+1} \ldots \tilde{\beta}_{k(X)}\tilde{\alpha}_{k(X)}\}\rceil$, and $\tilde{\alpha}_j$ is an arbitrary total order of all synthesized attributes, which are the maximal elements of the relation $E(X)\lceil A(X) \setminus \{a \mid a \in A(X), a \text{ occurs in } \tilde{\beta}_{j+1}\tilde{\alpha}_{j+1} \ldots \tilde{\beta}_{k(X)}\tilde{\alpha}_{k(X)}\}\rceil$. □

Intuitively, we are interested in the *existence* of an evaluation order. Therefore, if there are different total orders derivable from an is/si-graph of a nonterminal symbol X, then we select one total order, refer to it as *the* evaluation order of X, and denote it by T_X.

Example 3.20 (continued). The total orders

$$\begin{array}{ll} T_Z = \tilde{\beta}_Z\tilde{\alpha}_Z & \text{with } \tilde{\beta}_Z = \varepsilon, \text{ and } \tilde{\alpha}_Z = \alpha_0 \\ T_L = \tilde{\beta}_L\tilde{\alpha}_L & \text{with } \tilde{\beta}_L = r, \text{ and } \tilde{\alpha}_L = lv \end{array}$$

are the evaluation orders of the nonterminal symbols Z and L, respectively. Another total order for the attributes of L is achieved by swapping the order of the attributes l and v, i.e., $T'_L = rvl$. (Due to the observation that $k(Z) = 1$ and $k(L) = 1$ holds, the notation is simplified by adopting the names of the nonterminal symbols for the indices.) ■

Now we are in the state to define the class of ordered mags, which is the subclass of noncircular mags.

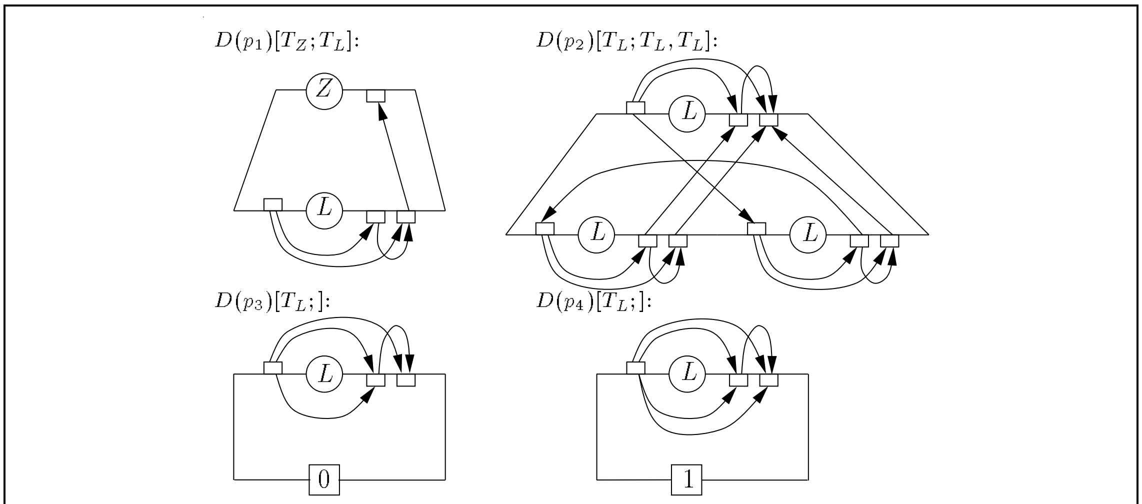


Figure 8: Dependency graphs extended by the evaluation orders T_Z und T_L.

Definition 3.21 (Ordered mag). An mag $M = (G_0, D, B, R)$ is called *ordered*, if for every nonterminal symbol X of G_0 an evaluation order T_X derived from $is/si(X)$ exists, such that for every production $p = X_0 \to w_0X_1w_1 \ldots X_nw_n$ of G_0, the graph $D(p)[T_{X_0}; T_{X_1}, \ldots, T_{X_n}]$ is acyclic. The *class of ordered mags* is denoted by $OMAG$. □

Example 3.22 (continued). In Figure 8 the dependency graphs extended by the derived evaluation orders T_Z and T_L are shown. Every graph in Figure 8 is noncircular, so the example mag is ordered. Thus, $M_{10} \in OMAG$. ∎

Note that there are mags, for which evaluation orders exist for every nonterminal symbol, but which are not ordered. Such an mag is given in Figure 9.
The class of ordered mags is a subclass of the class of noncircular mags.

Lemma 3.23. $OMAG \subset NCMAG$.

Proof. (1) The inclusion $OMAG \subseteq NCMAG$ is shown by a contradiction. Assume that M is an mag with $M \in OMAG$ and $M \notin NCMAG$. Then there is at least one derivation tree with a circular dependency graph. Since the is/si-graphs of the nonterminal symbols of G_0 are the union of all dependencies between the instances of the attributes of the nonterminal symbols, which occur in a derivation tree, at least one nonterminal symbol X with a *circular* is/si*-graph* exists and *no* T_X can be derived. Therefore $M \notin OMAG$ holds.
(2) Figure 9 shows a noncircular, but not ordered mag. Thus, $NCMAG \backslash OMAG \neq \emptyset$. □

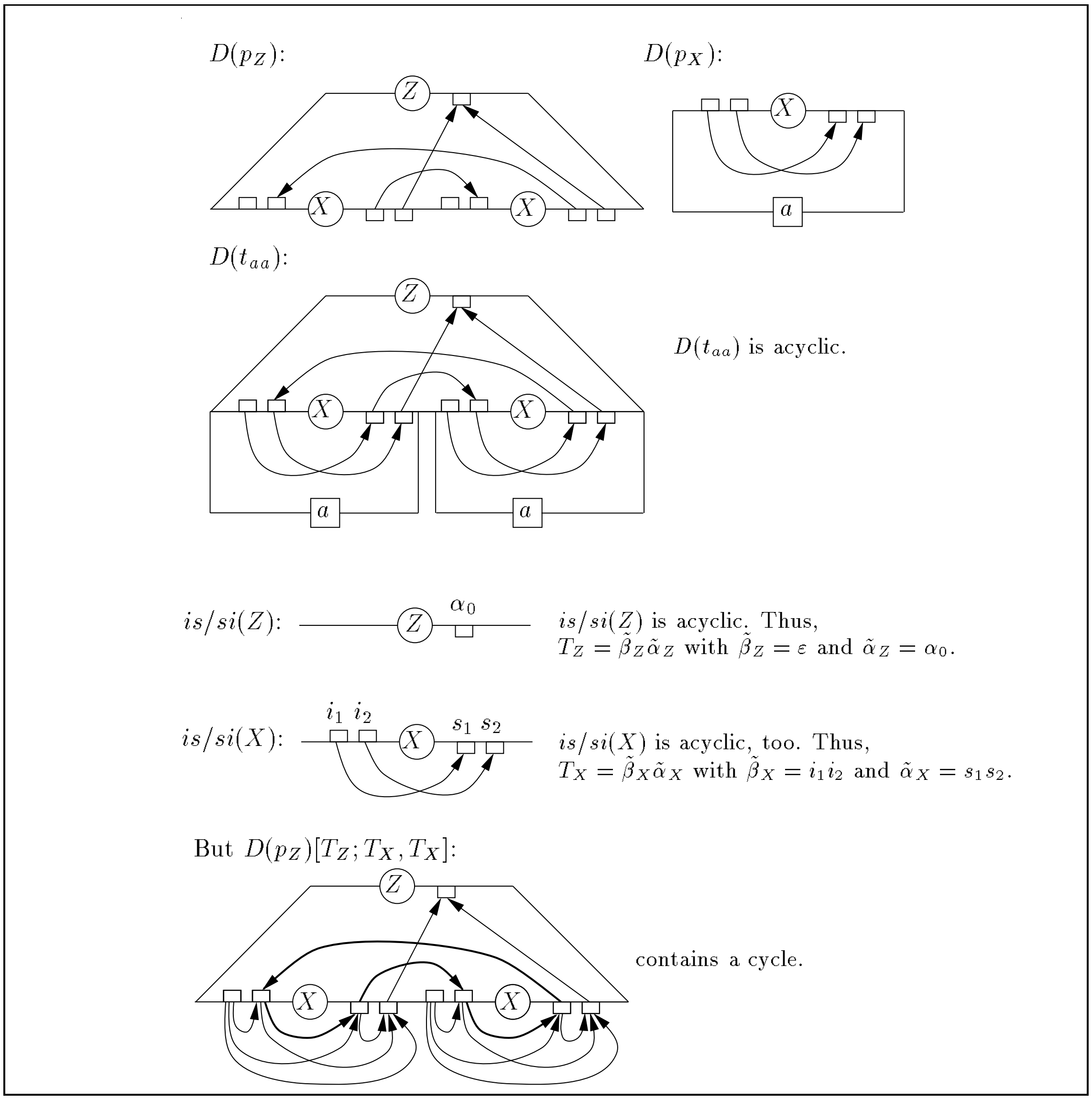


Figure 9: A noncircular mag, which is not ordered.

4 Coarse-grain evaluator for ordered macro attribute grammars

We develop two evaluators for ordered mags—an iterative evaluator and an incremental evaluator. Both evaluators use a term representation of the values of attribute instances, which we introduce first. Both evaluators apply a tree-walk mechanism to compute the admissible decoration of a given derivation tree, where the selection of nodes and the selection of attribute instances are controlled by *plans*. We derive the plans from the dependency graphs of the productions and the evaluation orders of the nonterminal symbols of a given ordered mag.

The iterative evaluator computes the admissible decoration of the given derivation tree by interpreting the plans starting at the root node of the derivation tree and stopping after every attribute instance was assigned a value.

The incremental evaluator computes the admissible decoration by interpreting the plans, too, but using the knowledge of a formerly computed admissible decoration, i.e., every derivation tree is considered to be the result of a modification of a formerly given derivation tree and the admissible decoration is updated in order to adapt it to the modified derivation tree.

For convenience, let M be an arbitrary ordered mag in the rest of our paper.

4.1 Term decoration of derivation trees

Intuitively, a tree-walk evaluator is an attribute evaluator which uses a given derivation tree as control structure to follow the dependency graph of the derivation tree. Visiting node by node the evaluator selects those attribute instances, whose direct predecessors in the dependency graph were already assigned a value. To compute the value of an attribute instance, a tree-walk evaluator uses the earlier computed values of the predecessing attribute instances, which therefore have to be stored. Since the attribute values are functions and the functions cannot be stored directly, a tree-walk evaluator has to store *representations* of the functions, such that the *interpretation* of the stored representations yields the desired mappings of the decoration val_t of t.

The representation of functions is realized by terms over function symbols and parameter symbols. To distinguish the decoration of a derivation tree from the representation of the decoration by terms, we introduce the *term decoration*.

Definition 4.1 (Term decoration). A *term decoration* $term_t$ *of* t is a family $(term_t^\tau \mid \tau \in \mathrm{K}^* \times \mathrm{K})$ of mappings $term_t^{(\kappa_1 \ldots \kappa_k, \kappa_0)} : A(t)^{(\kappa_1 \ldots \kappa_k, \kappa_0)} \to T_\Phi(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\kappa_0}$ with $k \geq 0$ and $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$. □

Example 4.2 (continued). The following family of mappings is a term decoration of the derivation tree t_{10} shown in Figure 5. Again, we abbreviate the parameter y^{nat} by y.

$$term_{t_{10}} = (term_{t_{10}}^{(\varepsilon, nat)}, term_{t_{10}}^{(nat, nat)}), \text{ with}$$

$$\begin{aligned}
term_{t_{10}}^{(nat,nat)}(\langle r,1\rangle) &= s(y), & term_{t_{10}}^{(nat,nat)}(\langle v,1211\rangle) &= y,\\
term_{t_{10}}^{(nat,nat)}(\langle r,12\rangle) &= s(y), & term_{t_{10}}^{(nat,nat)}(\langle l,121\rangle) &= s^4(y),\\
term_{t_{10}}^{(nat,nat)}(\langle r,122\rangle) &= s(y), & term_{t_{10}}^{(nat,nat)}(\langle v,121\rangle) &= s(s(y)),\\
term_{t_{10}}^{(nat,nat)}(\langle l,122\rangle) &= s(y), & term_{t_{10}}^{(nat,nat)}(\langle l,12\rangle) &= s^4(y),\\
term_{t_{10}}^{(nat,nat)}(\langle v,122\rangle) &= y, & term_{t_{10}}^{(nat,nat)}(\langle v,12\rangle) &= s(s(y)),\\
term_{t_{10}}^{(nat,nat)}(\langle r,121\rangle) &= s(s(y)), & term_{t_{10}}^{(nat,nat)}(\langle r,11\rangle) &= s^8(y),\\
term_{t_{10}}^{(nat,nat)}(\langle r,1212\rangle) &= s(s(y)), & term_{t_{10}}^{(nat,nat)}(\langle l,11\rangle) &= s^8(y),\\
term_{t_{10}}^{(nat,nat)}(\langle l,1212\rangle) &= s(s(y)), & term_{t_{10}}^{(nat,nat)}(\langle v,11\rangle) &= s^8(y),\\
term_{t_{10}}^{(nat,nat)}(\langle v,1212\rangle) &= s(s(y)), & term_{t_{10}}^{(nat,nat)}(\langle l,1\rangle) &= s^8(y),\\
term_{t_{10}}^{(nat,nat)}(\langle r,1211\rangle) &= s^4(y), & term_{t_{10}}^{(nat,nat)}(\langle v,1\rangle) &= s^{10}(y),\\
term_{t_{10}}^{(nat,nat)}(\langle l,1211\rangle) &= s^4(y), \quad\text{and} & term_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0,\varepsilon\rangle) &= s^{10}(null)
\end{aligned}$$

■

Clearly, we have to apply the idea of the admissibility of a decoration to a term decoration, too.

Definition 4.3 (Admissible term decoration). A term decoration $term_t$ is called *admissible*, if for every semantic equation $\langle a, x.i\rangle(y_1^{\kappa_1},\dots,y_k^{\kappa_k}) = [\zeta_{\langle a,i\rangle}]_x$ with $sort(\langle a,x.i\rangle) = (\kappa_1\dots\kappa_k,\kappa_0)$, $k\geq 0$, and $\kappa_0,\kappa_1,\dots,\kappa_k\in \mathrm{K}$ the equality

$$term_t^{(\kappa_1\dots\kappa_k,\kappa_0)}(\langle a,x.i\rangle) = \Xi^{(\kappa_1\dots\kappa_k,\kappa_0)}([\zeta_{\langle a,i\rangle}]_x)$$

holds, i.e., the two terms are equal. The mapping $\Xi^{(\kappa_1\dots\kappa_k,\kappa_0)}$ is a member of the family $(\Xi^\tau \mid \tau\in \mathrm{K}^*\times \mathrm{K})$ of mappings $\Xi^{(\kappa_1\dots\kappa_k,\kappa_0)} : T_{\Phi\cup A(t)}(\{y_1^{\kappa_1},\dots,y_k^{\kappa_k}\})^{\kappa_0}\to T_\Phi(\{y_1^{\kappa_1},\dots,y_k^{\kappa_k}\})^{\kappa_0}$, where the members of this family are defined simultaneously on the structure of their argument. Let $\tau = (\kappa_1\dots\kappa_k,\kappa_0)\in \mathrm{K}^*\times\mathrm{K}$ with $k\geq 0$ and $\kappa_0,\kappa_1,\dots,\kappa_k\in\mathrm{K}$. For $z\in T_{\Phi\cup A(t)}(\{y_1^{\kappa_1},\dots,y_k^{\kappa_k}\})^{\kappa_0}$ define

$\Xi^\tau(z) = z$, if $z = y_i^{\kappa_i}$ for some $i\in\{1,\dots,k\}$.

$= f(\Xi^{(\kappa_1\dots\kappa_k,\nu_1)}(z_1),\dots,\Xi^{(\kappa_1\dots\kappa_k,\nu_l)}(z_l))$,
if $z = f(z_1,\dots,z_l)$ for some $f\in\Phi^{(\nu_1\dots\nu_l,\kappa_0)}$ with $l\geq 0,\nu_1,\dots,\nu_l\in\mathrm{K}$,
and $z_i\in T_{\Phi\cup A(t)}(\{y_1^{\kappa_1},\dots,y_k^{\kappa_k}\})^{\nu_i}$ for every $i\in\{1,\dots,l\}$.

$= term_t^{(\nu_1\dots\nu_l,\kappa_0)}(\langle b,x.j\rangle)[y_1^{\nu_1} := \Xi^{(\kappa_1\dots\kappa_k,\nu_1)}(z_1),\dots,y_l^{\nu_l} := \Xi^{(\kappa_1\dots\kappa_k,\nu_l)}(z_l)]$,
if $z = \langle b,x.j\rangle(z_1,\dots,z_l)$ for some $\langle b,x.j\rangle\in A(t)^{(\nu_1\dots\nu_l,\kappa_0)}$ with $l\geq 0$,
$\nu_1,\dots,\nu_l\in\mathrm{K}$, and $z_i\in T_{\Phi\cup A(t)}(\{y_1^{\kappa_1},\dots,y_k^{\kappa_k}\})^{\nu_i}$ for every $i\in\{1,\dots,l\}$. □

Again, the noncircularity of an mag guarantees, that there is exactly one admissible term decoration for every derivation tree of the grammar.

Example 4.4 (continued). The term decoration given in Example 4.2 is the admissible term decoration of the derivation tree t_{10}.

In order to demonstrate the admissibility of $term_{t_{10}}$, we calculate for the semantic equation $\langle \alpha_0, \varepsilon\rangle = \langle v, 1\rangle(null)$ the value of $\Xi^{(\varepsilon,nat)}(\langle v,1\rangle(null))$.

$$\begin{aligned}\Xi^{(\varepsilon,nat)}(\langle v,1\rangle(null)) &= term_{t_{10}}^{(nat,nat)}(\langle v,1\rangle)[y := \Xi^{(\varepsilon,nat)}(null)]\\ &= term_{t_{10}}^{(nat,nat)}(\langle v,1\rangle)[y := null]\\ &= s^{10}(y)[y := null],\\ &= s^{10}(null).\end{aligned}$$

Thus, $term_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0,\varepsilon\rangle) = \Xi^{(\varepsilon,nat)}(\langle v,1\rangle(null))$. The interpretation (cf. Definition 2.4) of the calculated term $term_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0,\varepsilon\rangle)$ yields

$$val_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0,\varepsilon\rangle) = \hat{\varphi}^{(\varepsilon,nat)}(s^{10}(null)) = 10.$$

The calculation according to the other semantic equations of t_{10} is similar. ■

The correspondence between the admissible decoration val_t and the admissible term decoration $term_t$ of every derivation tree t, as it is stated in Lemma 4.5 below, assures the representation of val_t as $term_t$ in our attribute evaluators.

Lemma 4.5. Let $M = (G_0, D, B, R) \in NCMAG$. Let $t \in DT(G_0)$ be a derivation tree, val_t be the admissible decoration, and $term_t$ be the admissible term decoration of t. Let $x \in node(t)$ with $label_t(x) \in N$ be a node of t. Then for every attribute instance $\langle a,x\rangle^\tau \in A(t)$ with $\tau \in \mathrm{K}^* \times \mathrm{K}$, the equality $val_t^\tau(\langle a,x\rangle) = \hat{\varphi}^\tau(term_t^\tau(\langle a,x\rangle))$ holds.

Proof. (Sketch) The main idea is to topologically sort the attribute instances according to the dependency graph. The existence of at least one topological sorting is guaranteed by the noncircularity of the mag. Then, for every attribute instance in the topological sorting, Lemma 4.5 can be proved by induction over the structure of the terms on the right hand side of the semantic equations, because $term_t$ and val_t are admissible. Three different structural constituents have to be considered: parameters, i.e., the K-sorted elements of Y, function symbols, i.e., the $(\mathrm{K}^* \times \mathrm{K})$-sorted elements of Φ, and the $(\mathrm{K}^* \times \mathrm{K})$-sorted set of attribute instances. The proof of the lemma for the first instance in the sequence of the sorted attribute instances is straightforward, because only parameters and function symbols can be found on the right hand side of its semantic equation. The first instance in the sequence is the base of an induction on the length of the sequence of sorted attribute instances: On the right hand side of the semantic equation of an arbitrary attribute instance only those attribute instances can be found, which occur earlier in the sequence. Now the induction can be realized by considering the occurring attribute instances as the third structural case (besides the parameters and function symbols). The complete proof can be found in [Jok96]. □

4.2 Iterative evaluator for the term decoration

Recall that we have already fixed an arbitrary evaluation order T_X for every $X \in N$.

In a natural way the evaluation orders of the nonterminal symbols derived from their *is/si*-graphs induce orders over subsets of the attribute occurrences of the productions: the set of attribute occurrences $\{\langle a, j\rangle \mid a \in A(X_j)\}$ belonging to the jth nonterminal symbol X_j of a production $X_0 \to w_0 X_1 \ldots X_n w_n$ takes over the evaluation order of the attributes of X_j. If T_{X_j} is the evaluation order of X_j with the form $\tilde{\beta}_1\tilde{\alpha}_1 \ldots \tilde{\beta}_{k(X_j)}\tilde{\alpha}_{k(X_j)}$, then denote for every $i \in \{1, \ldots, k(X_j)\}$ the *suborder of attribute occurrences* which corresponds to the suborder $\tilde{\alpha}_i$ (and $\tilde{\beta}_i$, respectively) by $\langle\tilde{\alpha}_i, j\rangle$ (and $\langle\tilde{\beta}_i, j\rangle$, respectively). The set of all suborders of attribute occurrences of p is denoted by $\tilde{A}(p)$.

Example 4.6 (continued). The following sets of suborders of attribute occurrences are derived from the productions p_1, p_2, p_3, and p_4 and from the evaluation orders T_Z and T_L.

$\tilde{A}(p_1) = \{\langle\tilde{\beta}_Z, 0\rangle, \langle\tilde{\alpha}_Z, 0\rangle, \langle\tilde{\beta}_L, 1\rangle, \langle\tilde{\alpha}_L, 1\rangle\}$,
where $\langle\tilde{\beta}_Z, 0\rangle = \varepsilon$, $\langle\tilde{\alpha}_Z, 0\rangle = \langle\alpha_0, 0\rangle$, $\langle\tilde{\beta}_L, 1\rangle = \langle r, 1\rangle$, and $\langle\tilde{\alpha}_L, 1\rangle = \langle l, 1\rangle\langle v, 1\rangle$.

$\tilde{A}(p_2) = \{\langle\tilde{\beta}_L, 0\rangle, \langle\tilde{\alpha}_L, 0\rangle, \langle\tilde{\beta}_L, 1\rangle, \langle\tilde{\alpha}_L, 1\rangle, \langle\tilde{\beta}_L, 2\rangle, \langle\tilde{\alpha}_L, 2\rangle\}$,
where $\langle\tilde{\beta}_L, 0\rangle = \langle r, 0\rangle$, $\langle\tilde{\alpha}_L, 0\rangle = \langle l, 0\rangle\langle v, 0\rangle$, $\langle\tilde{\beta}_L, 1\rangle = \langle r, 1\rangle$, $\langle\tilde{\alpha}_L, 1\rangle = \langle l, 1\rangle\langle v, 1\rangle$,
$\langle\tilde{\beta}_L, 2\rangle = \langle r, 2\rangle$, and $\langle\tilde{\alpha}_L, 2\rangle = \langle l, 2\rangle\langle v, 2\rangle$.

$\tilde{A}(p_3) = \{\langle\tilde{\beta}_L, 0\rangle, \langle\tilde{\alpha}_L, 0\rangle\}$, where $\langle\tilde{\beta}_L, 0\rangle = \langle r, 0\rangle$ and $\langle\tilde{\alpha}_L, 0\rangle = \langle l, 0\rangle\langle v, 0\rangle$.

$\tilde{A}(p_4) = \{\langle\tilde{\beta}_L, 0\rangle, \langle\tilde{\alpha}_L, 0\rangle\}$, where $\langle\tilde{\beta}_L, 0\rangle = \langle r, 0\rangle$ and $\langle\tilde{\alpha}_L, 0\rangle = \langle l, 0\rangle\langle v, 0\rangle$. ■

Definition 4.7 (Evaluation graph of a production). Let $p = X_0 \to w_0 X_1 w_1 \ldots X_n w_n$ be a production of G_0. Let $D(p) = (A(p), E(p))$ be the dependency graph of p. The *evaluation graph of* p, denoted by $\tilde{D}(p)$, is the graph $\tilde{D}(p) = (\tilde{A}(p), \tilde{E}(p))$ over the set $\tilde{A}(p)$ of suborders of attribute occurrences of p with the edge relation $\tilde{E}(p) = \tilde{E}_{solid}(p) \cup \tilde{E}_{dashed}(p) \cup \tilde{E}_{dotted}(p)$, where:

$$\begin{aligned}
\tilde{E}_{solid}(p) = \{(\langle\tilde{\gamma}_{i_1}, j_1\rangle, \langle\tilde{\gamma}_{i_2}, j_2\rangle) \mid\ & j_1, j_2 \in \{0, \ldots, n\}, 1 \le i_1 \le k(X_{j_1}), 1 \le i_2 \le k(X_{j_2}),\\
& \langle\tilde{\gamma}_{i_1}, j_1\rangle \text{ is a suborder of outer attribute occurrences,}\\
& \langle\tilde{\gamma}_{i_2}, j_2\rangle \text{ is a suborder of inner attribute occurrences,}\\
& \text{and an edge } (\langle c_1, j_1\rangle, \langle c_2, j_2\rangle) \in E(p) \text{ exists,}\\
& \text{such that } \langle c_1, j_1\rangle \text{ occurs in } \langle\tilde{\gamma}_{i_1}, j_1\rangle \text{ and } \langle c_2, j_2\rangle \text{ occurs in } \langle\tilde{\gamma}_{i_2}, j_2\rangle\}\\
\tilde{E}_{dashed}(p) = \{(\langle\tilde{\beta}_i, j\rangle, \langle\tilde{\alpha}_i, j\rangle) \mid\ & j \in \{0, \ldots, n\}, i \in \{1, \ldots, k(X_j)\}\}\\
\cup \{(\langle\tilde{\alpha}_i, j\rangle, \langle\tilde{\beta}_{i+1}, j\rangle) \mid\ & j \in \{0, \ldots, n\}, i \in \{1, \ldots, k(X_j) - 1\}\}\\
\tilde{E}_{dotted}(p) = \{(\langle\tilde{\beta}_1, 0\rangle, \langle\tilde{\beta}_1, j\rangle) \mid\ & j \in \{1, \ldots, n\}\}\\
\cup \{(\langle\tilde{\alpha}_{k(X_j)}, j\rangle, \langle\tilde{\alpha}_{k(X_0)}, 0\rangle) \mid\ & j \in \{1, \ldots, n\}\}
\end{aligned}$$
□

Intuitively, $\tilde{E}_{solid}(p)$ contains the edges induced by the dependency graph of p itself. $\tilde{E}_{dashed}(p)$ contains the edges induced by the evaluation orders. $\tilde{E}_{dotted}(p)$ contains the edges induced by the traversing strategy, i.e., the instances of the attribute occurrences in $\langle\tilde{\beta}_1, j\rangle$ with $j \ge 1$ cannot be reached by the evaluator *before* the instances of the occurrences in $\langle\tilde{\beta}_1, 0\rangle$, and the instances of the occurrences in $\langle\tilde{\alpha}_{k(X_j)}, j\rangle$ with $j \ge 1$ cannot be reached by the evaluator *after* the instances of the occurrences in $\langle\tilde{\alpha}_{k(X_0)}, 0\rangle$. Note that the relations $\tilde{E}_{solid}(p)$, $\tilde{E}_{dashed}(p)$, and $\tilde{E}_{dotted}(p)$ may overlap.

The concept of evaluation graphs is closely related to the concept of *partition graphs* introduced by Engelfriet and Filé in [EF82]. The two concepts differ in two points. Firstly, the vertices of an evaluation graph are suborders of attribute occurrences, which are induced by the evaluation orders of the nonterminal symbols, while the partition graphs of [EF82] are built over partitions of attribute occurrences. Secondly, the subset $\tilde{E}_{dotted}$ of the edge relation is introduced within the evaluation graphs, which is not specified for the partition graphs. The edges in $\tilde{E}_{dotted}(p)$ explicitly reflect the tree-walk strategy, while the approach based on the partition graphs implicitly uses the tree-walk strategy in the definition and construction of the evaluation orders.

Example 4.8. The evaluation graphs constructed for the productions of the example mag M_{10} are shown in Figure 10, where the relations $\tilde{E}_{solid}(p)$, $\tilde{E}_{dashed}(p)$, and $\tilde{E}_{dotted}(p)$ are denoted by solid, dashed, and dotted arrows, respectively. ■

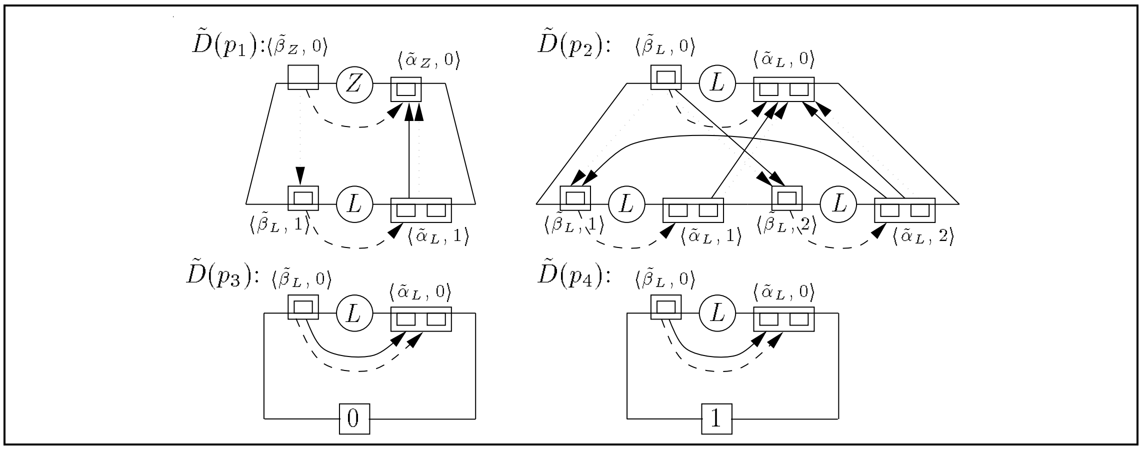


Figure 10: Evaluation graphs of the productions p_1, p_2, p_3, and p_4.

The evaluation graphs of the productions are the base for the construction of the *evaluation orders* of the productions.

Definition 4.9 (Evaluation order of a production). Let $p \in P$ be a production. Let $\tilde{D}(p) = (\tilde{A}(p), \tilde{E}(p))$ be the evaluation graph of p. An *evaluation order of p derived from* $\tilde{D}(p)$ is a topological sorting of $\tilde{D}(p)$. □

It is possible that for one evaluation graph different topological sortings exist. Then every topological sorting is an appropriate evaluation order of the production. But for the construction of the iterative evaluator we need only one evaluation order for every production of the grammar. Hence, we fix an arbitrary evaluation order for every $p \in P$ and call it T_p.

Example 4.10 (continued). The orders

$$\begin{aligned}
T_{p_1} &= \langle\tilde{\beta}_Z, 0\rangle\langle\tilde{\beta}_L, 1\rangle\langle\tilde{\alpha}_L, 1\rangle\langle\tilde{\alpha}_Z, 0\rangle, \\
T_{p_2} &= \langle\tilde{\beta}_L, 0\rangle\langle\tilde{\beta}_L, 2\rangle\langle\tilde{\alpha}_L, 2\rangle\langle\tilde{\beta}_L, 1\rangle\langle\tilde{\alpha}_L, 1\rangle\langle\tilde{\alpha}_L, 0\rangle, \\
T_{p_3} &= \langle\tilde{\beta}_L, 0\rangle\langle\tilde{\alpha}_L, 0\rangle, \text{ and} \\
T_{p_4} &= \langle\tilde{\beta}_L, 0\rangle\langle\tilde{\alpha}_L, 0\rangle
\end{aligned}$$

are evaluation orders of the productions p_1, p_2, p_3, and p_4. Other topological sortings of the evaluation graphs do not exist. ■

The evaluation orders of the productions are coded as *plans* of the productions.

Definition 4.11 (Plan of a production). Let $p = X_0 \rightarrow w_0X_1w_1 \ldots X_nw_n$ be a production of G_0. Let $T_p = \tilde{o}_1\tilde{o}_2 \ldots \tilde{o}_m$ with $m = |\tilde{A}(p)|$ be the evaluation order of p. The *plan of* p, denoted by $Plan(p)$, is the sequence of instructions $Plan(p) = instruction_1 \ldots instruction_m Suspend(k(X_0))$, where for every $l \in \{1, \ldots, m\}$, $instruction_l$ is a sequence of the following form:

$instruction_l = Visit(j, i)$, if $\tilde{o}_l = \langle \tilde{\alpha}_i, j\rangle$ for some suborder $\langle \tilde{\alpha}_i, j\rangle$ of outer attribute occurrences of p, such that $1 \leq j \leq n$ and $1 \leq i \leq k(X_j)$.

$= Suspend(i-1)$, if $\tilde{o}_l = \langle \tilde{\beta}_i, 0\rangle$ for some suborder $\langle \tilde{\beta}_i, 0\rangle$ of outer attribute occurrences of p, such that $1 \leq i \leq k(X_0)$.

$= Eval(\langle c_1, j\rangle) \ldots Eval(\langle c_r, j\rangle)$, if $\tilde{o}_l = \langle \tilde{\gamma}_i, j\rangle$ for some suborder $\langle \tilde{\gamma}_i, j\rangle = \langle c_1, j\rangle \ldots \langle c_r, j\rangle$ of inner attribute occurrences of p, such that $0 \leq j \leq n$, $1 \leq i \leq k(X_j)$, $r \geq 0$, and $c_\nu \in A(X_j)$ for every $1 \leq \nu \leq r$. □

The plans of the productions are used to control the traversal of the tree, i.e., to determine which node has to be visited next, and to select the attribute instance, which has to be assigned a value next.

The selection of an attribute instance is rather simple: if the evaluator has focussed the node x and an $Eval(\langle a, j\rangle)$-instruction would have to be executed, then just the attribute instance $\langle a, x.j\rangle$ has to be selected.
Let us have a closer look to the $Visit$- and $Suspend$-instructions occurring in the plans of two productions $p = prod_t(x_p)$ and $q = prod_t(x_q)$ applied at two neighbour nodes x_p and x_q, respectively.

$Plan(p):\ \ldots\ Visit(j,i)\ Instruction_{p,Visit(j,i)}\ \ldots$

$Plan(q):\ \ldots\ Suspend(i-1)\ Instruction_{q,Suspend(i-1)}\ \ldots\ Suspend(i)\ \ldots,$

Figure 11: The interleaving of plans.

If we assume, that x_q is the jth successor of x_p, then the ith visit to x_q, with $i \in \{1, \ldots, k(label_t(x_q))\}$, is triggered by a $Visit(j,i)$-instruction occurring in the plan of p. Now the evaluator interrupts the execution of $Plan(p)$ and reactivates the plan $Plan(q)$ by executing the instruction occurring in $Plan(q)$ immediately after the $Suspend(i-1)$-instruction, here denoted by $Instruction_{q,Suspend(i-1)}$, because this was the point, at which the plan $Plan(q)$ was suspended at the preceding visit to x_q. Then the evaluator executes every instruction, which follows after $Instruction_{q,Suspend(i-1)}$, until a $Suspend(i)$-instruction occurs. This is the point at which $Plan(q)$ has to be suspended again and at which $Plan(p)$ has to be reactivated. This plan has to be continued by executing the instruction which occurs immediately after the $Visit(j,i)$-instruction, here denoted by $Instruction_{p,Visit(j,i)}$, because this was the point, at which the execution of $Plan(p)$ was interrupted due to the visit of node x_q. The interleaved execution of the plans $Plan(p)$ and $Plan(q)$ is illustrated in Figure 11. Note in the definition of $Plan(p)$ that if $\tilde{o}_l = \langle \tilde{\beta}_1, 0 \rangle$, then $instruction_l = Suspend(0)$. Clearly, this instruction cannot be executed by the evaluator, because $Suspend(0)$ denotes a suspend, which has to be executed, *before the first visit* to a subtree. It can be removed from the sequence without changing the behavior of the evaluator.

Instead of searching the reactivation points in the plans at the runtime of the evaluator, the evaluator uses the mapping $MapToPlanIndex : P \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, which maps every tuple consisting of

1. a production p, which is applied at a new current node x,
2. a position of a nonterminal symbol in p:
   - if $Plan(p)$ is reactivated by a $Visit$-instruction, then the number 0
   - if $Plan(p)$ is reactivated by a $Suspend$-instruction in the plan of the jth successor of x, then the number j
3. a number i of a visit,

to an index of an instruction occurring in $Plan(p)$.

The mapping *MapToPlanIndex* can be constructed at the time the plans are constructed.

Example 4.12 (continued). From the evaluation orders T_{p_1}, T_{p_2}, T_{p_3}, and T_{p_4} the following plans and the mapping MapToPlanIndex are constructed.

$$\begin{aligned}
Plan(p_1) &= Eval(\langle r,1\rangle)\ Visit(1,1)\ Eval(\langle \alpha_0,0\rangle)\ Suspend(1)\\
Plan(p_2) &= Eval(\langle r,2\rangle)\ Visit(2,1)\ Eval(\langle r,1\rangle)\ Visit(1,1)\\
&\quad\ Eval(\langle l,0\rangle)\ Eval(\langle v,0\rangle)\ Suspend(1)\\
Plan(p_3) &= Eval(\langle l,0\rangle)\ Eval(\langle v,0\rangle)\ Suspend(1)\\
Plan(p_4) &= Eval(\langle l,0\rangle)\ Eval(\langle v,0\rangle)\ Suspend(1)
\end{aligned}$$

$$\begin{aligned}
&MapToPlanIndex(p_1,0,1) = 1\\
&MapToPlanIndex(p_1,1,1) = 3\\
&MapToPlanIndex(p_2,0,1) = 1\\
&MapToPlanIndex(p_2,2,1) = 3\\
&MapToPlanIndex(p_2,1,1) = 5\\
&MapToPlanIndex(p_3,0,1) = 1\\
&MapToPlanIndex(p_4,0,1) = 1
\end{aligned}$$

■

Additionally, a mapping is needed which computes for two neighbour nodes x and $x.$-1 the successor number of x with respect to the preceding node $x.$-1. To obtain this number, we define the mapping $SuccessorNo$ as follows. If $x = x.\text{-}1.j$ with $j > 0$, i.e., the node x is the jth successor of its predecessor $x.$-1, then $SuccessorNo(x, x.\text{-}1) = j$, otherwise $SuccessorNo(x, x.\text{-}1)$ is undefined.

The *iterative evaluator for ordered mags* is given by Algorithm 4.13. It uses the procedures $DoVisit$, $DoSuspend$, and $DoEval$ which are shown in separate figures.

Algorithm 4.13 (Iterative evaluator).

Input : a derivation tree $t \in DT(G_0)$,
the mapping $MapToPlanIndex$ and for every production p a plan $Plan(p)$,

Output : an admissible term decoration $term_t$ of t,

Variables: x holds the node activated at the beginning of the **repeat** -loop,
$PlanIndex$ holds the index of the plan-instruction to be executed next,
j holds the number of a successor node,
r holds the number of a visit,
$Instruction$ holds an instruction of a plan,
$ready \in$ **boolean** is a flag used to detect the end of the evaluation.

(1) **begin**
(2) $\quad x := root(t)$;
(3) $\quad PlanIndex := 1$;
(4) $\quad ready :=$ **false** ;
(5) $\quad$ **repeat**
(6) $\quad\quad Instruction := Plan[prod_t(x)][PlanIndex]$;
(7) $\quad\quad$ **case** $Instruction$ **of**
(8) $\quad\quad\quad Visit(j, r) \;:\; DoVisit(j, r) \mid$
(9) $\quad\quad\quad Suspend(r) \;:\; DoSuspend(r) \mid$
(10) $\quad\quad\quad Eval(\langle a, j\rangle) : DoEval(\langle a, j\rangle)$
(11) $\quad\quad$ **end**
(12) $\quad$ **until** $ready$
(13) **end** □

The iterative evaluator is an interpreter of the plans of the productions. The main algorithm (lines 1, ..., 13 in Algorithm 4.13) initializes the traversal (lines 2, 3, and 4) and starts the main control loop (lines 5, ..., 12), which simply fetches an instruction from the current plan (line 6) and calls the executing procedure (lines 7, ..., 11).

(1) **procedure** $DoVisit(j, r)$;
(2) **begin**
(3) $\quad PlanIndex := MapToPlanIndex(prod_t(x.j), 0, r)$;
(4) $\quad x := x.j$
(5) **end**

Figure 12: $DoVisit$ procedure of the iterative evaluator.

The *DoVisit*- and the *DoSuspend*-procedures, which are shown in Figure 12 and 13, respectively, have to switch the currently visited node (line 4 and 6, respectively) and the currently used plan (line 3 and 5, respectively). The position at which the execution of a reactivated plan has to be continued is determined by the *MapToPlanIndex*-function explained above. Additionally, in the *DoSuspend*-procedure the end of the traversal is detected (line 3), because it is determined by the attempt to execute a *Suspend*-instruction at the root node of the given derivation tree.

procedure $DoSuspend(r)$;
begin
    if $x = root(t)$ **then** $ready :=$ **true**
    else
        $PlanIndex := MapToPlanIndex(prod_t(x.\text{-}1), SuccessorNo(x, x.\text{-}1), r)$;
        $x := x.\text{-}1$
    end
end

Figure 13: *DoSuspend* procedure of the iterative evaluator.

(1) **procedure** $DoEval(\langle a, j\rangle)$;
(2) **begin**
(3)   Let $(\langle a, j\rangle(y_1, \ldots, y_k) = \zeta_{\langle a,j\rangle}) \in R(prod_t(x))$, where $sort(\langle a, j\rangle) = \tau$;
(4) $term_t^{\tau}(\langle a, x.j\rangle) := \Xi^{\tau}([\zeta_{\langle a,j\rangle}]_x)$;
(5) $PlanIndex := PlanIndex + 1$
(6) **end**

Figure 14: *DoEval* procedure of the iterative evaluator.

The *DoEval*-procedure is shown in Figure 14. It computes the term representation of the value of an attribute instance by applying the Ξ-mapping introduced in Definition 4.3 to the right hand side of the semantic equation (line 4). Then, the instruction, which immediately follows the *Eval*-instruction, has to be fetched. Therefore the *PlanIndex* is increased (line 5).

4.3 Incremental evaluator for the term decoration

The iterative evaluator for ordered mags computes the admissible term decoration of a derivation tree t. After each modification of this tree t the decoration of the modified tree t' has to be changed in order to make it consistent with the new tree structure. The substitution of a subtree of t at a node x' of t by a tree s' is illustrated in Figure 15.

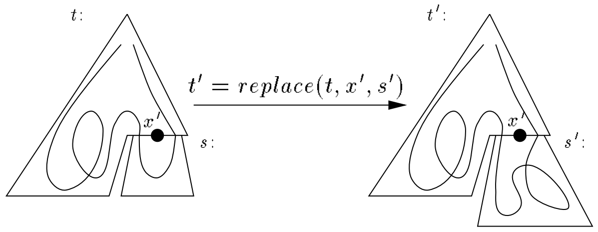


Figure 15: Substitution of a subtree s of a derivation tree t by a tree s'.

Of course, it were possible to employ the iterative evaluator to do the job. But since we expect only a subtree of a given derivation tree to be replaced by another subtree, intuition tells that some parts of the derivation tree and of its decoration are not affected by the substitution. Since the iterative evaluator starts at the root and computes the value of *every* attribute instance, it would traverse unchanged regions of the tree and would perform evaluations of attribute instances, whose values are already correct due to the new decoration. The process can be optimized by an *incremental* evaluation algorithm. We restrict the set of attribute instances which are assigned a value after a subtree substitution to those instances which belong to the *affected area*, i.e., which are either new or changed, because they depend on affected attribute instances. We simply denote the elements of the affected area by *affected instances.* In [DRT83], [Yeh83] and [RT89], the idea of affected area was applied to optimize the evaluation of ags. Intuitively, the affected area depends on the node of a derivation tree, at which the subtree has been substituted, and can be derived from the dependency graph of the derivation tree.

Definition 4.14 (Affected area). Let $t \in DT(G_0)$ be a derivation tree. Let $x' \in node(t)$ with $X = label_t(x')$ and $X \in N$. x' is called *node of the subtree replacement.* Let s' be an X-derivation tree. Let $t' = replace(t, x', s')$ be a modified tree. Let $D(t') = (A(t'), E(t'))$ be the dependency graph of t'.

The *area affected by the substitution of s' at x'*, denoted by $A_{affected}$, is the subset $A_{affected} \subseteq A(t')$ of attribute instances of t', defined as follows: $A_{affected} = A_{new} \cup A_{changed}$, where $A_{new} = \{\langle a, x\rangle \mid \langle a, x\rangle \in A(t'), x = x'v \text{ for some } v \in I\!N_+^+\}$ and

$$A_{changed} = \{\langle a, x\rangle \mid \langle a, x\rangle \in A(t'), x \neq x'v \text{ for every } v \in I\!N_+^+, \text{ there is } (\langle s, x'\rangle, \langle a, x\rangle) \in E(t')^* \text{ with } s \in S\text{-}Att, val_{t'}(\langle a, x\rangle) \neq val_t(\langle a, x\rangle)\}. \quad \square$$

A very similar characterization of the affected area used for an incremental evaluator of ordered ags was given by Yeh in [Yeh83]. The sets AFFECT, NEWBORN, and NOTEQU, described in [Yeh83], correspond to the sets $A_{affected}$, A_{new}, and $A_{changed}$, respectively. We observe, that the set of *new* attribute instances and the set of instances whose values have *changed* are strictly disjoint, i.e., $A_{new} \cap A_{changed} = \emptyset$.

Let us illustrate Definition 4.14 by applying it to our example.

Example 4.15 (continued). In Figure 16 the derivation tree t_{14} is shown, which can be derived from the derivation tree t_{10} of Figure 5 by the following substitution step: In the derivation tree t_{10} the subtree with the root node $x' = 1211$ is replaced by the L-derivation tree, which is emphasized by dashed lines in Figure 16. This substitution induces the affected area $A_{affected} = A_{new} \cup A_{changed}$ with

$$A_{new} = \emptyset,$$
$$A_{changed} = \{\langle v, 1211\rangle, \langle v, 121\rangle, \langle v, 12\rangle, \langle v, 1\rangle, \langle \alpha_0, \varepsilon\rangle\}.$$

In Figure 16 the members of the affected area are marked by the symbol $\bullet$. The instance $\langle l, 1211\rangle$, which is marked by the symbol $\circ$, does not belong to the affected area, because it is not new and it has not changed its value. ■

There is a slight difference between the definition of $A_{affected}$ and the use of the variable $A_{affected}$ in the evaluator of Algorithm 4.16. Since the evaluator does not know the members of the affected area in advance, it has to make some *assumption* about the membership

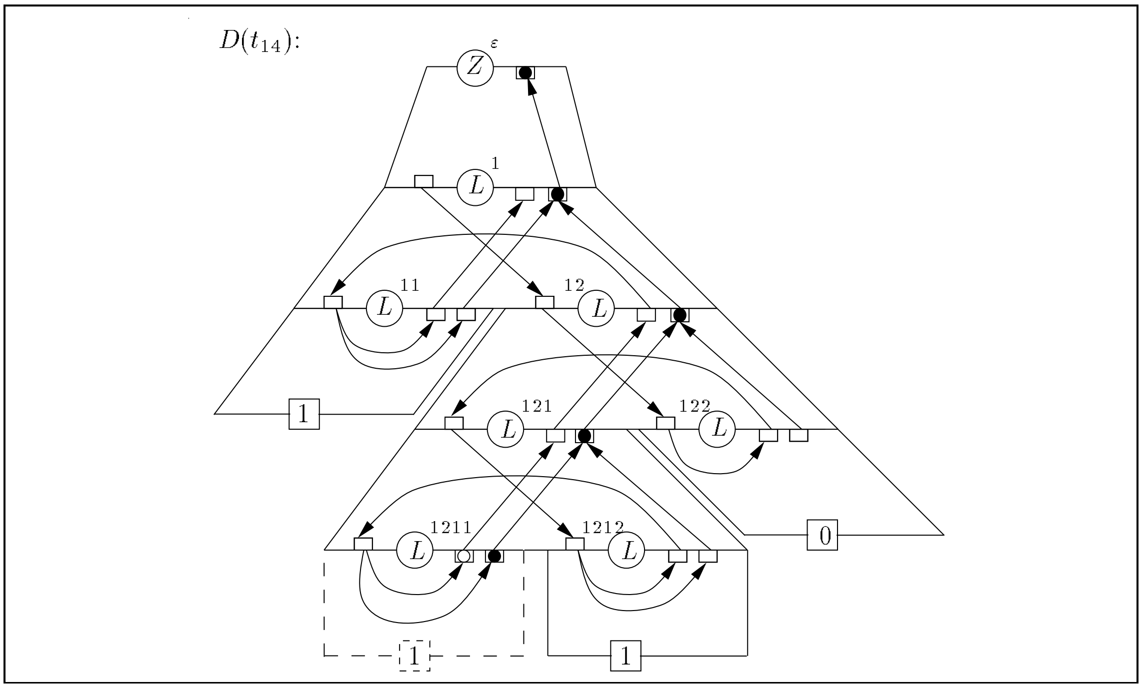


Figure 16: Dependency graph of the derivation tree t_{14} of the word 1110.

of the instances, which has to be checked later. Therefore the variable $A_{affected}$ of the algorithm is initialized with the set of all new instances and of *all* synthesized attribute instances of the node of the subtree replacement. Thus, for every affected instance, which is not new, it is tested, whether its value has changed or not. In what follows, every instance, whose value has changed, induces an update of the variable $A_{affected}$ by adding all instances, which depend on the changed one, because their values *may have changed*, too.

Algorithm 4.16 (Incremental evaluator).

Input : a derivation tree $t' \in DT(G_0)$ with $D(t') = (A(t'), E(t'))$,
which is the result of a substitution step,
the mapping $MapToPlanIndex$ and for every production p a plan $Plan(p)$,
the node x' of the subtree replacement,
the term decoration $term_t$, which was admissible immediately
before the subtree replacement.

Output : an admissible term decoration $term_{t'}$ of t'.

Variables: x holds the node activated at the beginning of the **repeat** -loop,
$PlanIndex$ holds the index of the plan-instruction to be executed next,
j, j' hold numbers of neighbour nodes,
r holds the number of a visit,
$Instruction$ holds an instruction of a plan,
$ready \in$ **boolean** is a flag used to detect the end of the evaluation,
$Neighbour$ holds the root node of a subtree
possibly owning some instances depending on a recomputed instance,
N_{new} holds the set of nodes of the new subtree,

$N_{affected}$ holds the set of nodes with affected instances,
$OldValue \in T_\Phi(Y)$ holds an existing term of an instance,
$A_{affected}$ holds the set of affected instances.

(1) **begin**
(2) $\quad N_{affected} := \{x \mid x \in node(t'), x = x'v$ for some $v \in I\!N_+^*\}$;
(3) $\quad N_{new} := N_{affected} \setminus \{x'\}$;
(4) $\quad A_{affected} := \{\langle a, x\rangle \mid \langle a, x\rangle \in A(t'), x \in N_{new}\}$
$\qquad\qquad \cup\{\langle s, x'\rangle \mid \langle s, x'\rangle \in A(x'), s \in S\text{-}Att\}$;
(5) $\quad$ **for each** $\langle a, x\rangle \in A(t') \setminus A_{affected}$ **do**
(6) $\qquad term_{t'}(\langle a, x\rangle) := term_t(\langle a, x\rangle)$
(7) $\quad x := x'$;
(8) $\quad PlanIndex := 1$;
(9) $\quad ready :=$ **false** ;
(10) $\quad$ **repeat**
(11) $\qquad Instruction := Plan[prod_{t'}(x)][PlanIndex]$;
(12) $\qquad$ **case** $Instruction$ **of**
(13) $\qquad\quad Visit(j, r) \;:\; DoVisit(j, r) \mid$
(14) $\qquad\quad Suspend(r) : DoSuspend(r) \mid$
(15) $\qquad\quad Eval(\langle a, j\rangle) : DoEval(\langle a, j\rangle)$
(16) $\qquad$ **end**
(17) $\quad$ **until** $ready$
(18) **end** □

procedure $DoVisit(j, r)$;
begin
$\quad$ **if** $x.j \in N_{affected}$ **then**
$\qquad PlanIndex := MapToPlanIndex(prod_{t'}(x.j), 0, r)$;
$\qquad x := x.j$
$\quad$ **else**
$\qquad PlanIndex := PlanIndex + 1$
$\quad$ **end**
end

Figure 17: $DoVisit$ procedure of the incremental evaluator.

The incremental evaluator takes over the structure of the iterative evaluator, i.e., it consists of the main algorithm (lines $1, \ldots, 18$), and it uses the procedures *DoVisit*, *DoSuspend* and *DoEval*, which are shown in Figure 17, 18, and 19, respectively. Additionally, the procedure *AddDependentInstances*, which is shown in Figure 20, updates the affected area due to the progress of the evaluator. The main algorithm of the incremental evaluator initializes the set of affected nodes (line 2), which is needed to control the traversal of the tree, and the set of affected instances (line 4). The value of every attribute instance of t', which is not an affected instance is adopted from the value of the

(1) **procedure** $DoSuspend(r)$;
(2) **begin**
(3) **if** $x = root(t')$ **then** $ready :=$ **true**
(4) **elsif** $x.\text{-}1 \in N_{affected}$ **then**
(5) $PlanIndex := MapToPlanIndex(prod_{t'}(x.\text{-}1), SuccessorNo(x, x.\text{-}1), r)$;
(6) $x := x.\text{-}1$
(7) **elsif** $r = k(label_{t'}(x))$ **then** $ready :=$ **true**
(8) **else** $PlanIndex := PlanIndex + 1$
(9) **end**
(10) **end**

Figure 18: $DoSuspend$ procedure of the incremental evaluator.

corresponding instance of t (line 5 and 6). The starting point of the traversal is the root node x' of the new subtree (line 7).

(1) **procedure** $DoEval(\langle a, j\rangle)$;
(2) **begin**
(3) **if** $\langle a, x.j\rangle \in A_{affected}$ **then**
(4) Let $(\langle a, j\rangle(y_1, \ldots, y_k) = \zeta_{\langle a,j\rangle}) \in R(prod_{t'}(x))$ with $sort(\langle a, j\rangle) = \tau$;
(5) **if** $x.j \in N_{new}$ **then** $(*\ \langle a, x.j\rangle \in A_{new}\ *)$
(6) $term^{\tau}_{t'}(\langle a, x.j\rangle) := \Xi^{\tau}([\zeta_{\langle a,j\rangle}]_x)$
(7) **else** $(*\ \langle a, x.j\rangle \in A_{changed}\ *)$
(8) $OldValue := term^{\tau}_{t'}(\langle a, x.j\rangle)$;
(9) $term^{\tau}_{t'}(\langle a, x.j\rangle) := \Xi^{\tau}([\zeta_{\langle a,j\rangle}]_x)$;
(10) **if** $term^{\tau}_{t'}(\langle a, x.j\rangle) \neq OldValue$ **then**
(11) $AddDependentInstances(\langle a, j\rangle)$
(12) **end**
(13) **end**
(14) **end**
(15) $PlanIndex := PlanIndex + 1$
(16) **end**

Figure 19: $DoEval$ procedure of the incremental evaluator.

The characteristic difference between the procedures of the iterative evaluator in the Figures 12, 13, and 14 and the corresponding procedures of the incremental evaluator in the Figures 17, 18, and 19, respectively, is that the former procedures are executed rather unconditionally, while the execution of the latter depends on the set of affected instances or on the set of affected nodes. Thus, a visit to a node is provided, only if it is the root node of a subtree which contains attribute instances which are affected (lines 3, 4, and 5 of

(1) **procedure** $AddDependentInstances(\langle a, j\rangle)$;
(2) **begin**
(3)   **if** $j \neq 0$ **then** $Neighbour := x.j$
(4)   **elsif** $x \neq root(t')$ **then** $Neighbour := x.\text{-}1$
(5)   **else** $Neighbour := NULL$
(6)   **end**
(7)   **if** $Neighbour \neq NULL$ **and** an instance $\langle a', Neighbour.j'\rangle \in A(t')$
        with $j' \geq 0$ and $(\langle a, x.j\rangle, \langle a', Neighbour.j'\rangle) \in E(t')$ exists **then**
(8) $N_{affected} := N_{affected} \cup \{Neighbour\}$;
(9)   **for each** $\langle a', Neighbour.j'\rangle \in A(t')$ with
                $(\langle a, x.j\rangle, \langle a', Neighbour.j'\rangle) \in E(t')$ **do**
(10) $A_{affected} := A_{affected} \cup \{\langle a', Neighbour.j'\rangle\}$
(11)   **end**
(12)   **end**
(13) **end**

Figure 20: *AddDependendInstances* procedure of the incremental evaluator.

Figure 17). In analogy, a plan is suspendend, only if the evaluator expects some changes "outside" the currently visited subtree (lines 4, 5, and 6 of Figure 18). The evaluation of an attribute instance (cf. Figure 19) is performed, only if it is in the set of affected instances (line 3). But now, the evaluator has to distinguish two cases (lines 5, ..., 13). The value of an instance in the *new* subtree has to be computed always and there is no value to which it can be compared (lines 5 and 6). The value of an affected instance which is not new has been computed already for the previous admissible decoration. Thus, the evaluator has to check, if the last modification of the derivation tree influences the value of the attribute instance. Therefore, the value of the affected instance is computed again (line 9) after its old value has been stored (line 8). If the evaluator recognizes that the old value and the new value differ (line 10), then the set of affected instances has to be updated by all instances, whose values depend on the new value (line 11).

The detection of the instances which depend on an attribute instance $\langle a, x.j\rangle$ is formulated in Figure 20. Since the evaluator computes the values of instances of inner attribute occurrences by using the values of instances of outer attribute occurrences, the evaluator has to look for instances which depend on $\langle a, x.j\rangle$ in a context in which $\langle a, x.j\rangle$ is the instance of an outer attribute occurrence. Thus, if the instance $\langle a, x.j\rangle$ is an inherited attribute instance, this context is the part of the dependency graph corresponding to the dependency graph of the production applied at $x.j$. In analogy, if the instance $\langle a, x.j\rangle$ is a synthesized attribute instance, this context is the part of the dependency graph corresponding to the dependency graph of the production applied at $x.-1$, if $x.-1$ exists. The right context is determined in the lines 3 through 6 in Figure 20 and coded as the value of the variable *Neighbour*.

If an attribute instance depending on $\langle a, x.j\rangle$ is detected, then the evaluator has to update the set of affected nodes in order to reach the node *Neighbour* during the traversal, and it has to add *all* instances which directly depend on $\langle a, x.j\rangle$ as new members to the affected area.

Example 4.17 (continued). Initially, the variable $A_{affected}$ contains the set $\{\langle l, 1211\rangle, \langle v, 1211\rangle\}$ of the synthesized attribute instances of the node x' (cf. Figure 16). In the *DoEval*-procedure the values of these instances are recomputed and compared to their *OldValues*. The value of the attribute instance $\langle l, 1211\rangle$ has not changed. Therefore, no update to the affected area has to be made. The value of the attribute instance $\langle v, 1211\rangle$ has changed. Therefore, the affected area is stepwise extended. First the instance $\langle v, 121\rangle$ is added to the affected area, then the instances $\langle v, 12\rangle$, $\langle v, 1\rangle$, and $\langle \alpha_0, \varepsilon\rangle$ are added. Of course they are recomputed by the evaluator. The following table shows the new admissible term decoration $term_{t_{14}} = (term_{t_{14}}^{(\varepsilon,nat)}, term_{t_{14}}^{(nat,nat)})$ in comparison to the old admissible term decoration $term_{t_{10}}$, where the parameter y^{nat} is abbreviated by y, and the instances are listed in the sequence, the evaluator picks them up.

$$
\begin{array}{rcl}
term_{t_{10}}^{(nat,nat)}(\langle l, 1211\rangle) = s^4(y) & = & term_{t_{14}}^{(nat,nat)}(\langle l, 1211\rangle) = s^4(y), \\
term_{t_{10}}^{(nat,nat)}(\langle v, 1211\rangle) = y & \neq & term_{t_{14}}^{(nat,nat)}(\langle v, 1211\rangle) = s^4(y), \\
term_{t_{10}}^{(nat,nat)}(\langle v, 121\rangle) = s(s(y)) & \neq & term_{t_{14}}^{(nat,nat)}(\langle v, 121\rangle) = s^6(y), \\
term_{t_{10}}^{(nat,nat)}(\langle v, 12\rangle) = s(s(y)) & \neq & term_{t_{14}}^{(nat,nat)}(\langle v, 12\rangle) = s^6(y), \\
term_{t_{10}}^{(nat,nat)}(\langle v, 1\rangle) = s^{10}(y) & \neq & term_{t_{14}}^{(nat,nat)}(\langle v, 1\rangle) = s^{14}(y), \\
\text{and } term_{t_{10}}^{(\varepsilon,nat)}(\langle \alpha_0, \varepsilon\rangle) = s^{10}(null) & \neq & term_{t_{14}}^{(\varepsilon,nat)}(\langle \alpha_0, \varepsilon\rangle) = s^{14}(null).
\end{array}
$$

The interpretation (cf. Definition 2.4) of the term $term_{t_{14}}^{(\varepsilon,nat)}(\langle \alpha_0, \varepsilon\rangle)$ yields

$$val_{t_{14}}^{(\varepsilon,nat)}(\langle \alpha_0, \varepsilon\rangle) = \hat{\varphi}^{(\varepsilon,nat)}(s^{14}(null)) = 14.$$

Thus, $\tau_{tv}(M_{10})(t_{14}) = 14$. ■

Observation 4.18 (Granularity of the evaluation). If the recomputation of the value of an attribute instance is required, the incremental evaluator recomputes the term of an attribute instance *in a whole* (cf. Figure 19, line 9). This is done, though only *parts* of the term structure may have changed. Here, the changed parts are copies of the terms of other attribute instances on which the recomputed attribute instance depends. □

To obtain a view to the values of attribute instances which allow a finer granularity for the manipulation of the values, we introduce a graph based approach for the evaluation of macro attribute grammars in the following section.

5 Fine-grain evaluator for ordered macro attribute grammars

We introduce an extension of the hyperedge substitution of [Hab92]. Our extension allows to keep the information which subgraphs of a hypergraph are the result of a hyperedge substitution. Note that this information is lost by the standard hyperedge substitution.

We define the graph decoration of a derivation tree, such that we are able to represent the values of attribute instances by graphs.

Based on the graph representation of the values of the attribute instances and based on the extended hyperedge substitution to update selected parts of the graphs of the attribute instances, we develop an incremental evaluator. Our approach combines the selective recomputation, which was realized in the incremental evaluator for the term decoration, with an incremental computation of the attribute values themselves, thus, yielding a fine-grain incremental method for ordered mags.

5.1 Extended parjungles and extended hyperedge substitution

The representation of terms by graphs and the realization of term rewriting by graph rewriting is based on the notion of parjungles and hyperedge substitution, respectively.

Recall that K is a finite, nonempty set of sorts. Also, in the rest of this paper let Σ and Γ be two disjoint $(\mathrm{K}^* \times \mathrm{K})$-sorted sets and let $Y_m = \{y_1, \ldots, y_m\}$ with $m \geq 0$ be a K-sorted set of parameters.

Definition 5.1 (Parjungle). A *parjungle* h *over* Σ *with* m *parameters* $y_1, \ldots, y_m$ is an acyclic hypergraph (cf. Definition 2.8) $(V_h, E_h, label_h, source_h, target_h, begin_h, end_h)$ with:

- V_h is a K-sorted set of vertices.
- E_h is a $(\mathrm{K}^* \times \mathrm{K})$-sorted set of hyperedges.
- For every $e \in E_h^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$ with $k \geq 0$ and $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$, $label_h(e) \in \Sigma^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$.
- For every $e \in E_h^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$ the incident vertices of e are $source_h(e) \in V_h^{\kappa_1} \circ \ldots \circ V_h^{\kappa_k}$ and $target_h(e) \in V_h^{\kappa_0}$ with the following properties.

  (1) For every $v \in V_h \backslash \{v \mid v \text{ occurs in } begin_h\}$, there is an $e \in E_h$ with $target_h(e) = v$, i.e., every nonparameter vertex is the target vertex of an edge.

  (2) For every $v \in V_h \cap \{v \mid v \text{ occurs in } begin_h\}$, there is no edge $e \in E_h$ with $target_h(e) = v$, i.e., no parameter vertex is the target vertex of an edge.

  (3) There are no edges $e, e' \in E_h$ with $e \neq e'$, such that $target_h(e) = target_h(e')$, i.e., edges do not share the same target vertex.

- The external vertices $begin_h = v_1 \ldots v_m \in {V_h}^*$ are the m *parameter vertices*, such that for every $i, j \in \{1, \ldots, m\}$, if $i \neq j$, then $v_i \neq v_j$, and for every $i \in \{1, \ldots, m\}$, $sort(v_i) = sort(y_i)$. The external vertex $end_h \in V_h$ is the unique *root vertex of* h.

□

The *set of parjungles over* Σ *with* m *parameters* is denoted by $J_\Sigma(Y_m)$. The union $\bigcup_{m\geq 0} J_\Sigma(Y_m)$ is denoted by $J_\Sigma(Y)$.

The application of parjungles in conjunction with hyperedge substitution as rewriting method would be suitable, if we were interested in computing the decoration of a single derivation tree. But this method does not support incremental recomputation of the resulting graphs. The reason is that standard hyperedge substitution applied to standard parjungles means *removing* an edge and *replacing* it by another parjungle. The recomputation of such a graph means *substituting* the earlier inserted parjungle by a new one. But the inserted parjungle can not be extracted from the whole graph, because the information, which vertices and edges belong to it, is lost.

To solve the problem we introduce *extended parjungles* to which *extended hyperedge substitution* is applied. The basic idea is that the "substituted" edge must not be removed, such that it is still present in the resulting hypergraph and therefore can be used to determine the inserted parjungle.

The extended parjungles, which are defined below in Definition 5.4, consist of "normal" hyperedges and *extended edges*.

Definition 5.2 (Extended edge). Let $g \in J_\Gamma(Y_m)$ be a parjungle over Γ with m parameters, such that $E_g = \{e\}$, and the incident vertices of $e \in E_g$ are $source_g(e) = begin_g$ and $target_g(e) = end_g$. Let $h \in J_\Sigma(Y_m) \cup \{interface(g)\}$. The identification of the external vertices of g and h yields the *extended edge* g'/I *over* Σ *and* Γ *(with* m *parameters)* which is the hypergraph with the following components. The set of vertices is $V_{g'} = V_g \cup V_h$. The set of edges is $E_{g'} = E_g \cup E_h$. The labeling function is $label_{g'} = (label_g \cup label_h)$. The incidence functions are $source_{g'} = (source_g \cup source_h)$ and $target_{g'} = (target_g \cup target_h)$. The external vertices are $begin_{g'} = begin_g$ and $end_{g'} = end_g$. The identification relation $I \subseteq V_{g'} \times V_{g'}$ is $I = \{(begin_g(i), begin_h(i)) \mid i \in \{1,\ldots,m\}\} \cup \{(end_g, end_h)\}$. □

The set of extended edges over Σ and Γ with m parameters is denoted by $\hat{J}_{\Sigma,\Gamma}(Y_m)$. The union $\bigcup_{m\geq 0} \hat{J}_{\Sigma,\Gamma}(Y_m)$ is denoted by $\hat{J}_{\Sigma,\Gamma}(Y)$. For every extended edge g' the subgraph h of g', by which the Γ-labeled edge e of g is extended, is called *subgraph spread by* e and is denoted by $spread(e)$. Note that $spread(e)$ is either a parjungle over Σ or the interface graph of g. The latter case reflects the situation, in which a substitution has not yet been peformed.

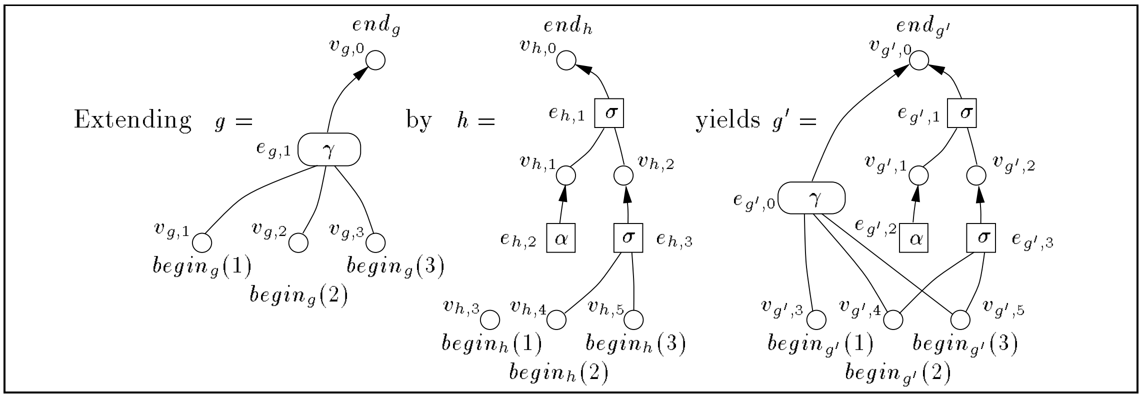


Figure 21: Construction of the extended edge g'.

Example 5.3 (Parjungle concepts). In Figure 21 the construction of an extended edge is illustrated. Let $K_1 = \{\kappa\}$ be a set of sorts. Let $\Gamma_1 = \{\gamma^{(\kappa\kappa\kappa,\kappa)}, \delta^{(\kappa\kappa,\kappa)}\}$ and $\Sigma_1 = \{\alpha^{(\varepsilon,\kappa)}, \sigma^{(\kappa\kappa,\kappa)}\}$ be two disjoint $({K_1}^* \times K_1)$-sorted sets of symbols.

The parjungle $g \in J_{\Gamma_1}(\{y_1^\kappa, y_2^\kappa, y_3^\kappa\})$ owns the edge $e_{g,1}$, which will be extended by the parjungle $h \in J_{\Sigma_1}(\{y_1^\kappa, y_2^\kappa, y_3^\kappa\})$, as it is shown in Figure 21, yielding the extended edge g'.

To express graphically the membership of an edge to the set of Σ-labeled edges and to the set of Γ-labeled edges, we use rectangular boxes and rectangular boxes with rounded corners, respectively. The vertices of a parjungle are drawn as circles. The circles of the source vertices of an edge are connected to the box of the edge by lines, where the left to right order of the lines reflects the order of the source vertices. The target vertex of an edge is connected by an arrow. ■

Definition 5.4 (Extended parjungle). Let $h \in J_{\Sigma\cup\Gamma}(Y_m)$ be a parjungle over $\Sigma \cup \Gamma$ with m parameters. For every Γ-labeled edge $e \in E_{h,\Gamma}^{(\kappa_1\ldots\kappa_k,\kappa_0)}$ let $g_e \in \hat{J}_{\Sigma,\Gamma}(Y_k)$ be an extended edge over Σ and Γ with $E_{g_e,\Gamma}=\{e'\}$ and $label_{g_e}(e') = label_h(e)$. An *extended parjungle over* Σ *and* Γ *with* m *parameters* is obtained from h by the substitution (cf. Definition 2.10) of every Γ-labeled edge $e \in E_{h,\Gamma}$ by its corresponding extended edge g_e. □

The *set of extended parjungles over* Σ *and* Γ *with* m *parameters* is denoted by $J_{\Sigma,\Gamma}(Y_m)$. The union $\bigcup_{m\geq 0} J_{\Sigma,\Gamma}(Y_m)$ is denoted by $J_{\Sigma,\Gamma}(Y)$.

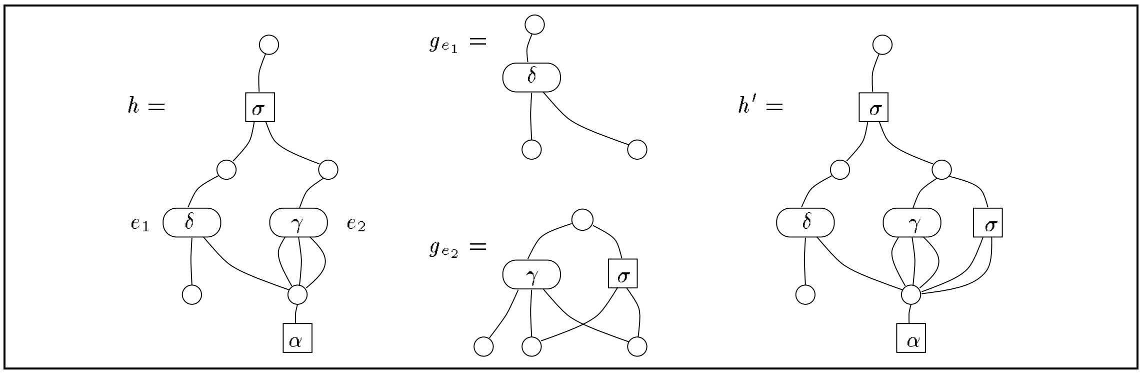


Figure 22: Construction of the extended parjungle h' over Σ_1 and Γ_1.

Example 5.5 (Parjungle concepts continued). In Figure 22 the extended parjungle h' over Σ_1 and Γ_1 with one parameter y_1^κ is shown on the right. It is constructed by substituting the Γ-labeled edges e_1 and e_2 of h, which is shown on the left, by the extended edges g_{e_1} and g_{e_2}, respectively. Note that g_{e_1} is an extended edge, where the subgraph spread by the Γ-labeled edge is the interface graph of g_{e_1} only.

The graphical notation of the parjungles is simplified as follows. Since every edge is directed bottom-up, we do not need arrows any longer to denote the direction of an edge. In what follows, we can arrange the unique root vertex and the parameter vertices at the top and at the bottom of a parjungle, respectively. Since two vertices can be distinguished by their unique position in the figure, we avoid naming a vertex explicitly, if we do not want to refer to it in the text. We proceed with edges in the same way. Thus, only the edges e_1 and e_2 of the parjungle h are denoted. ■

Our intended substitution allows to replace a subgraph g in an extended parjungle h, which is an extended edge, by an extended edge g', if the Γ-labeled edges of g' and g have the same label.

Example 5.6. In Figure 23 the extended parjungle h'' over Σ_1 and Γ_1 with one parameter y_1^{κ} is shown. h'' is the extended parjungle which is obtained by substituting the extended edge g_{e_2} of h' by the extended edge g'_{e_2}. ■

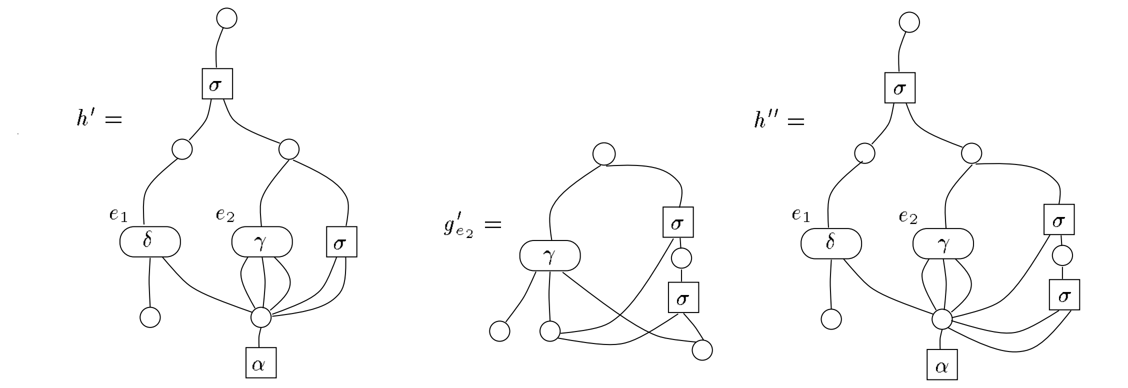


Figure 23: Substitution of the extended edge g_{e_2} in h' by an extended edge g'_{e_2}.

Note that an extended edge g, which has to be substituted, is fully determined by the Γ-labeled edge e of g. Therefore we view the substitution of g as the substitution of e and the subgraph spread by e, which is called *extended hyperedge substitution.*

Definition 5.7 (Extended hyperedge substitution). Let $h \in J_{\Sigma,\Gamma}(Y_m)$ be an extended parjungle over Σ and Γ with m parameters. Let $e \in E_{h,\Gamma}$ with $label_h(e) = \gamma^{(\kappa_1 \dots \kappa_l, \kappa_0)}$, with $l \geq 0$ and $\kappa_0, \dots, \kappa_l \in \mathrm{K}$. Let $g \in \hat{J}_{\Sigma,\Gamma}(\{y_1^{\kappa_1}, \dots, y_l^{\kappa_l}\})^{\kappa_0}$ be an extended edge with $E_{g,\Gamma} = \{e'\}$ and $label_g(e') = label_h(e)$. The *extended hyperedge substitution of e and the subgraph spread by e in h by g*, yields the hypergraph $f/I_{e,g}$ with the following components. The set of vertices is $V_f = (V_h \setminus (V_{spread(e)} \setminus V_{interface(spread(e))})) \cup V_g$. The set of edges is $E_f = ((E_h \setminus \{e\}) \setminus E_{spread(e)}) \cup E_g$. The labeling function is $label_f = (label_h \cup label_g)\big|_{E_f}$. The incidence functions are $source_f = (source_h \cup source_g)\big|_{E_f}$ and $target_f = (target_h \cup target_g)\big|_{E_f}$. The external vertices are $begin_f = begin_h$ and $end_f = end_h$. The identification relation $I_{e,g} \subseteq V_f \times V_f$ induced by the substitution is $I_{e,g} = \{(source_h(e)(i), begin_g(i)) \mid i \in \{1, \dots, |\, source_h(e)\,|\}\} \cup \{(target_h(e), end_g)\}$. □

Note that the extended hyperedge substitution restores the Γ-labeled edge. Thus, the effect of the whole operation is the substitution of an older subgraph spread by the Γ-labeled edge by a new subgraph spread by the same edge.

Let $h \in J_{\Sigma,\Gamma}(Y)$ be an extended parjungle over Σ and Γ, let $\gamma \in \Gamma$, and let $g \in \hat{J}_{\Sigma,\Gamma}(Y)$ with $E_{g,\Gamma} = \{e_g\}$ be an extended edge, where $label_g(e_g) = \gamma$ and $spread(e_g){=}g'$ with $g' \in J_{\Sigma}(Y)$. The extended hyperedge substitution of every extended edge of h, which contains a hyperedge with label γ, by a new copy of g is denoted by $h[\gamma \Leftarrow g']$.
Let $\Gamma' \subseteq \Gamma$ and for every $\gamma \in \Gamma'$ let $g_\gamma \in \hat{J}_{\Sigma,\Gamma}(Y)$ with $E_{g_\gamma,\Gamma} = \{e_{g_\gamma}\}$, $label_{g_\gamma}(e_{g_\gamma}) = \gamma$, and $spread(e_{g_\gamma}) = g'_\gamma$ with $g'_\gamma \in J_{\Sigma}(Y)$. The extended hyperedge substitution of every

extended edge of h, which contains a hyperedge with label γ for some $\gamma \in \Gamma'$, by a new copy of g_γ is denoted by $h[\gamma \Leftarrow g'_\gamma \mid \gamma \in \Gamma]$.

The following observation pays attention to a node identification problem which occurs during the extended hyperedge substitution for a special case of extended edges.

Observation 5.8 (Cycle introduction by node identification). Let $g \in J_\Gamma(Y_m)$ be a parjungle over Γ with $E_g = \{e\}$, $source_g(e) = begin_g$, and $target_g(e) = end_g$. Let $h \in J_\Sigma(Y_m)$ with $E_h = \emptyset$. Thus, h consists of m distinct vertices $v_1, \ldots, v_m$, and the vertex end_h has to occur in $begin_h$. The extension of g by the parjungle h yields a hypergraph g', which has a vertex occurring in $begin_{g'}$, which is identified with the vertex $end_{g'}$. Thus, g' contains a cycle. Figure 24 demonstrates this problem. It follows, that

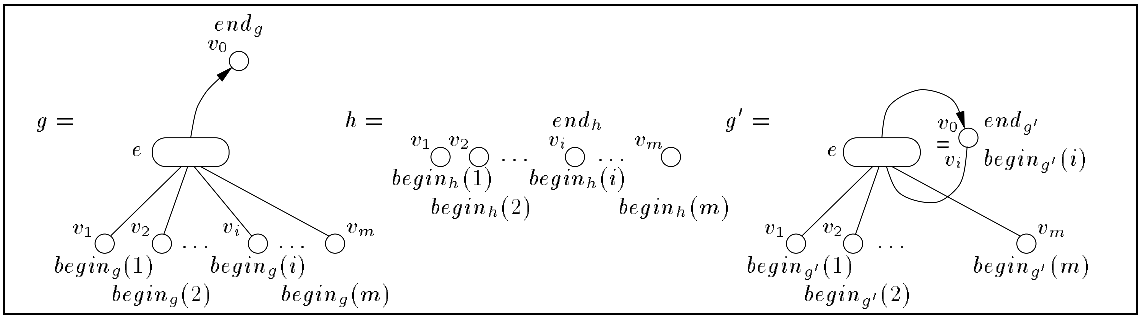


Figure 24: Illustration of the identification problem.

every extended parjungle over Σ and Γ, which is the result of an extended hyperedge substitution, where an extended edge is replaced by g', inherits the cycle in g'. These cycles cannot be splitted by further extended hyperedge substitutions. To avoid such an identification of the target vertex of an edge with one of the source vertices of the edge, we introduce a new $(\mathrm{K} \times \mathrm{K})$-sorted set $\mathcal{E}$ of symbols, where $\mathcal{E} \cap \Sigma = \emptyset$ and $\mathcal{E} \cap \Gamma = \emptyset$ holds:

$$\mathcal{E} = \{\varepsilon_\kappa^{(\kappa,\kappa)} \mid \kappa \in \mathrm{K}\}.$$

Figure 25 demonstrates the solution of the explained identification problem. □

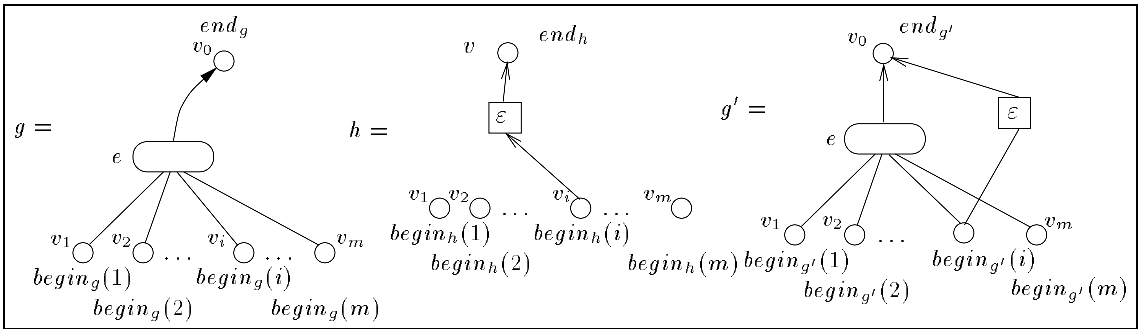


Figure 25: Solution of the identification problem.

5.2 Graph decoration of derivation trees

For the graphical representation of the values of the attribute instances, which are terms over function symbols and parameter symbols, as extended parjungles and for the calculation of trees from their graphical representations, we need two constructions. The first (Construction 5.9) produces the graph representation of a term, the second (Construction 5.10) produces the term representation of a graph.

The first construction of an extended parjungle which represents a given tree is based on the construction introduced within the proof of Lemma 5.5 of [EV94].

Construction 5.9 (Graph representation of a term). Let $k \geq 0$, $\kappa_0, \ldots, \kappa_k \in \mathrm{K}$, and $t \in T_{\Sigma\cup\Gamma}(\{y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}\})^{\kappa_0}$. Let $\mathcal{E} = \{\varepsilon_\kappa^{(\kappa,\kappa)} \mid \kappa \in \mathrm{K}\}$.
The *parjungle over* $\Sigma \cup \mathcal{E} \cup \Gamma$ *representing* t, denoted by $graph^{(\kappa_1\ldots\kappa_k,\kappa_0)}(t)$, is the hypergraph $(V, E, label, source, target, begin, end)$ with the following components. We must distinguish two cases for the structure of t:

1. If for every $i \in \{1, \ldots, k\}$, $t \neq y_i$, then
   - $V = \{v_s \mid s \in sub(t) \cup \{y_1, \ldots, y_k\}\}$ (Intuitively, for every parameter $y \in \{y_1, \ldots, y_k\}$, which is *not* a subtree of t, we have to construct the vertex v_y explicitly. Note, that V is the base for the sharing of equal subtrees. The reason is, that if the same subtree s occurs twice in t, only one vertex v_s is added to V.)
   - $E = \{e_s \mid s \in sub(t) \setminus \{y_1, \ldots, y_k\}\}$ (Since every parameter is represented as an external vertex, we do not construct an edge for it.)
   - For every $e_s \in E$ with $s = \sigma(s_1, \ldots, s_m)$ for some $\sigma \in (\Sigma\cup\Gamma)^{(\nu_1\ldots\nu_m,\nu_0)}$ with $m \geq 0$, $\nu_0, \ldots, \nu_m \in \mathrm{K}$, and $s_1, \ldots, s_m \in T_{\Sigma\cup\Gamma}(\{y_1, \ldots, y_k\})$, we have $label(e_s) = \sigma$, $source(e_s) = v_{s_1} \ldots v_{s_m}$, and $target(e_s) = v_s$.
   - $begin = v_{y_1} \ldots v_{y_k}$, and $end = v_t$.
2. If for some $i \in \{1, \ldots, k\}$, $t = y_i$, then $V = \{v, v_{y_1}, \ldots, v_{y_k}\}$, $E = \{e\}$, $label(e) = \varepsilon_{\kappa_0}^{(\kappa_0,\kappa_0)}$, $source(e) = v_{y_i}$, $target(e) = v$, $begin = v_{y_1} \ldots v_{y_k}$, and $end = v$. □

Intuitively, an extended parjungle h over $(\Sigma \cup \mathcal{E})$ and Γ can represent a tree over Σ. If every Γ-labeled edge of h, incident with a vertex on a path in h which starts with the root vertex of h, spreads a subgraph which contains at least one edge (i.e. it is not an interface graph or a parjungle without edges), then a tree over Σ can be obtained from h, if all Γ-labeled edges are ignored, and the remaining parjungle is interpreted as a tree.
The construction of the represented tree over Σ is formalized as follows. Note that Σ-labeled and $\mathcal{E}$-labeled edges are preferred and Γ-labeled edges are not considered.

Construction 5.10 (Represented tree). Let $h \in J_{\Sigma\cup\mathcal{E},\Gamma}(Y_m)$. If for every $e \in E_{h,\Gamma}$ with $target_h(e) = v$, such that v occurs on a path $v_0 e_1 v_1 \ldots e_m v_m$ with $v_0 = end_h$ and $m \geq 0$, the subgraph spread by e is a parjungle over $\Sigma \cup \mathcal{E}$ with at least one edge, then the *tree in* $T_\Sigma(Y_m)$ *represented by* h, denoted by $tree_\Sigma(h)$, is the tree $unfold_\Sigma(end_h)$, where the mapping $unfold_\Sigma : V_h \to T_\Sigma(Y_m)$, is recursively defined by

1. If for some $i \in \{1, \ldots, m\}$, $v = begin_h(i)$, then $unfold_\Sigma(v) = y_i$.

2. If for some $e \in E_{h,\Sigma\cup\mathcal{E}}$ with $label_h(e) = \varepsilon_\kappa \in \mathcal{E}$, $\kappa \in \mathrm{K}$, $source_h(e) = v'$, and $v = target_h(e)$, then $unfold_\Sigma(v) = unfold_\Sigma(v')$, i.e., $\mathcal{E}$-labeled edges are ignored.

3. If for some $e \in E_{h,\Sigma\cup\mathcal{E}}$ with $label_h(e) = \sigma \in \Sigma^{(\kappa_1\ldots\kappa_k,\kappa_0)}$, $k \geq 0$, $\kappa_0,\ldots,\kappa_k \in \mathrm{K}$, $source_h(e) = v_1\ldots v_k \in V_h{}^*$, and $v = target_h(e)$, then $unfold_\Sigma(v) = \sigma(unfold_\Sigma(v_1), \ldots, unfold_\Sigma(v_k))$. □

Note that for every tree $t \in T_\Sigma(Y)$, $tree_\Sigma(graph(t)) = t$. But for every extended parjungle $h \in J_{\Sigma\cup\mathcal{E},\Gamma}(Y)$, the parjungle $graph(tree_\Sigma(h))$ can be smaller than the original hypergraph h. The first reason is that the application of $tree_\Sigma(h)$ removes vertices and edges, which are not connected to the root vertex of h by a path. Additionally the application of $graph$ to the result of $tree_\Sigma(h)$ constructs a parjungle with maximal sharing of common subtrees, such that subgraphs of h, which were isomorphic but not shared, are now identified in $graph(tree_\Sigma(h))$.

The graphical representation of the values of the attribute instances is realized by extended parjungles over $\Phi\cup\mathcal{E}$ and $A(t)$, where Φ is the set of function symbols in the semantic domain, and $A(t)$ is the set of attribute instances of the derivation tree t. Thus, our disjoint sets Σ and Γ are instantiated by Φ and $A(t)$, respectively. The approach is formalized by the *graph decoration of t*.

Definition 5.11 (Graph decoration). A *graph decoration $graph_t$ of t* is a family $(graph_t^\tau \mid \tau \in \mathrm{K}^*\times\mathrm{K})$ of mappings

$$graph_t^{(\kappa_1\ldots\kappa_k,\kappa_0)} : A(t)^{(\kappa_1\ldots\kappa_k,\kappa_0)} \to J_{\Phi\cup\mathcal{E},A(t)}(\{y_1^{\kappa_1},\ldots,y_k^{\kappa_k}\})^{\kappa_0}$$

with $k \geq 0$ and $\kappa_0,\ldots,\kappa_k \in \mathrm{K}$. □

Definition 5.12 (Admissible graph decoration). A graph decoration $graph_t$ is called *admissible*, if for every semantic equation $\langle a, x.i\rangle(y_1^{\kappa_1},\ldots,y_k^{\kappa_k}) = [\zeta_{\langle a,i\rangle}]_x$ with $sort(\langle a, x.i\rangle) = \tau_a$ where $\tau_a = (\kappa_1\ldots\kappa_k,\kappa_0) \in \mathrm{K}^*\times\mathrm{K}$ the equality

$$\begin{aligned} graph_t^{\tau_a}(\langle a, x.i\rangle) = graph^{\tau_a}([\zeta_{\langle a,i\rangle}]_x)[&\langle b, x.j\rangle \Leftarrow graph^{\tau_b}(tree_\Phi(graph_t^{\tau_b}(\langle b, x.j\rangle))) \\ &\mid \langle b, x.j\rangle^{\tau_b} \in sym([\zeta_{\langle a,i\rangle}]_x)\cap A(t), \tau_b \in \mathrm{K}^*\times\mathrm{K}] \end{aligned}$$

holds. □

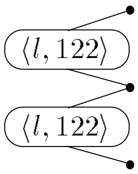
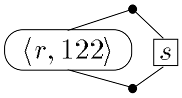
Note that in the right-hand side of our equation the subsequent applications of $tree_\Phi$ and $graph^{\tau_b}$ to $graph_t^{\tau_b}(\ldots)$ ignore the $A(t)$-labeled edges and can perform the optimization of parjungles which was mentioned above.

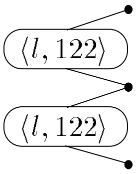
Example 5.13 (Continuation of Example 4.17). Let us explain the computation of the graph representation of the attribute instance $\langle r, 121\rangle$ of t_{10}. The following semantic equation defines the attribute instance: $\langle r, 121\rangle(y) = \langle l, 122\rangle(\langle l, 122\rangle(y))$.
Thus,

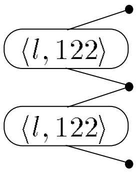
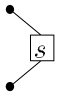
$$\begin{aligned} graph_{t_{10}}^{(nat,nat)}(\langle r, 121\rangle) &= graph^{(nat,nat)}(\langle l, 122\rangle(\langle l, 122\rangle(y))) \\ &\qquad [\langle l, 122\rangle \Leftarrow graph^{(nat,nat)}(tree_\Phi(graph_{t_{10}}^{(nat,nat)}(\langle l, 122\rangle)))]. \end{aligned}$$

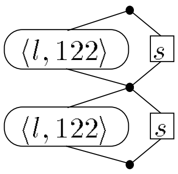
We assume that $graph_{t_{10}}^{(nat,nat)}(\langle l, 122\rangle)$ has already been computed. Hence,

$graph_{t_{10}}^{(nat,nat)}(\langle r, 121\rangle)$

$=$  $[\langle l, 122\rangle \Leftarrow graph^{(nat,nat)}(tree_{\Phi}($  $))]$

$=$  $[\langle l, 122\rangle \Leftarrow graph^{(nat,nat)}(s(y_1))]$

$=$  $[\langle l, 122\rangle \Leftarrow$  $]$

$=$ 

In Figure 26 the graph decoration of the derivation tree t_{10} is shown. ■

Lemma 5.14. Let $M = (G_0, D, B, R) \in NCMAG$. Let $t \in DT(G_0)$ be a derivation tree, val_t be the admissible decoration, and $graph_t$ be the admissible graph decoration of t. Let $x \in node(t)$ with $label_t(x) \in N$ be a node of t. Then for every attribute instance $\langle a, x\rangle^{\tau} \in A(t)$ with $\tau \in \mathrm{K}^* \times \mathrm{K}$, the equality $val_t^{\tau}(\langle a, x\rangle) = \hat{\varphi}^{\tau}(tree_{\Phi}(graph_t^{\tau}(\langle a, x\rangle)))$ holds.

Proof. (Sketch) To prove Lemma 5.14 we have to do the same steps as in the proof of Lemma 4.5: After topologically sorting the attribute instances of t according to the dependency graph of t, Lemma 5.14 is proved by induction over the structure of the terms on the right hand sides of the semantic equations. □

5.3 Incremental evaluator for the graph decoration

The incremental evaluator for the term decoration of a derivation tree (cf. Algorithm 4.16) is the foundation for our fine-grain incremental evaluator for the graph decoration of a derivation tree. Note the difference of the data structure between the evaluators: in Algorithm 4.16 the input and the output is a term decoration, whereas the new evaluator gets a graph decoration and delivers a graph decoration of the modified tree. The variable $OldValue$ is an extended parjungle over $\Phi \cup \mathcal{E}$ and $A(t)$, i.e. $OldValue \in J_{\Phi\cup\mathcal{E},A(t)}(Y)$.

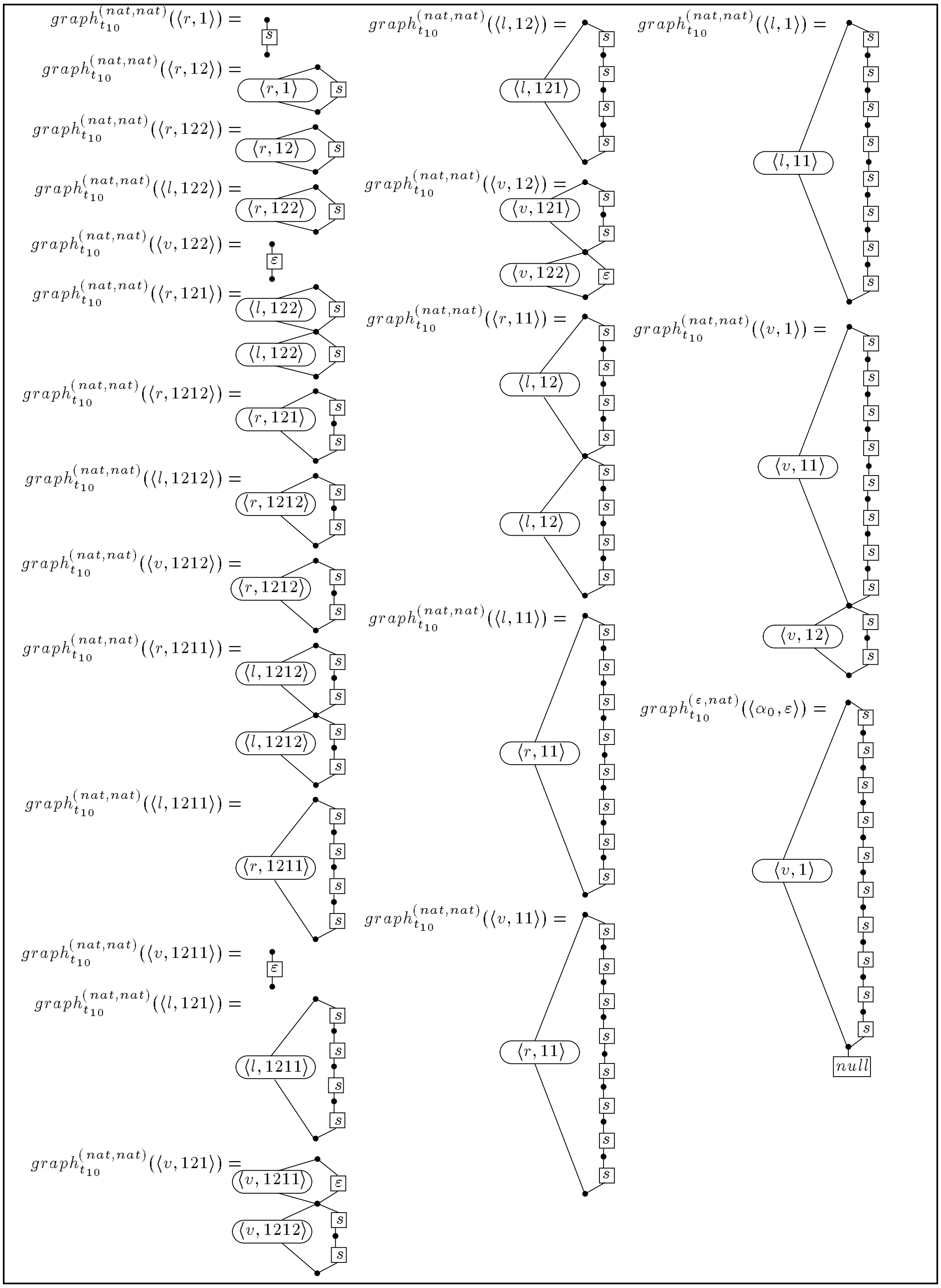


Figure 26: Graph decoration of the derivation tree t_{10}.

Since the new, fine-grain incremental evaluator optimizes the computation of the representation of the attribute values but not the selection of the attribute instances, whose values have to be recomputed, we can use the plans constructed for the incremental evaluator. In fact the result of the optimization is a modified $DoEval$-procedure. The main algorithm and the procedures $DoVisit$, $DoSuspend$, and $AddDependentInstances$ in Figure 17, Figure 18, and Figure 20, respectively, are left unchanged.

The optimized $DoEval$-procedure is shown in Figure 27. The main structure of the *DoEval*-procedure of the fine-grain incremental evaluator is a repetition of the main structure of the *DoEval*-procedure of the coarse-grain incremental evaluator in Figure 19. The reason is that this structure implements the selective recomputation, which is hardly influenced by the fine-grain view to the values of the instances. Thus, the statements which implement the following ideas can be found in both algorithms.

- The recomputation is performed only on affected attribute instances (compare lines 3, 37, and 38 in Figure 27 with lines 3, 14, and 15 in Figure 19).
- *New* and *changed* affected instances are handled in different ways (lines 5 and 11/23 in Figure 27 and lines 5 and 7 in Figure 19).
- The values of new instances are computed straightforward (line 6, ..., 10 in Figure 27 and line 6 in Figure 19).
- The value of an instance, which may have changed, i.e., which is affected but not new, is first recomputed (compare the two blocks in Figure 27 consisting of lines 12, ...,17 and lines 24, ..., 30, respectively, with lines 8 and 9 in Figure 19) and then the recomputed value is compared to the earlier computed value (lines 18, ..., 22 and lines 31, ..., 35 in Figure 27 and lines 10, ..., 12 in Figure 19), which possibly triggers the extension of the affected area (lines 19 and 32 in Figure 27 and line 11 in Figure 19).

To understand the differences between the coarse-grain evaluator and the fine-grain evaluator, let us have a closer look on how the latter computes the graph representation of the value of an attribute instance $\langle a, x.j\rangle \in A(t')$.

If the instance $\langle a, x.j\rangle$ is *new*, the graph representation has to be computed for the first time. Therefore the evaluator constructs an initial graph value (see line 6 in Figure 27) from the semantic equation for $\langle a, x.j\rangle$. After this is done, every edge of the resulting graph which is labeled by an attribute instance is substituted by the extended edge representing the current graph value of that instance (lines 7, ..., 10). To obtain the representation of the value of an instance $\langle a', x.j'\rangle$, the edges which are labeled by attribute instances have to be removed from the graph of $\langle a', x.j'\rangle$. This is formalized by applying first $tree_{\Phi}$ and then $graph^{\tau'}$ to the graph of $\langle a', x.j'\rangle$ (line 9). An efficient implementation of *DoEval* would rather remove the edges instead of constructing the tree and of transforming it again into a graph.

If the instance $\langle a, x.j\rangle$ has *changed*, then the evaluator has to distinguish between those instances which are synthesized at the node x' of subtree replacement and those which are not (lines 11 and 23). The reason is that the semantic equation which determines the value of a synthesized instance at x' has changed. Thus, it is not possible to compute the new graph by updating the old graph. The old graph and the new graph can have

procedure $DoEval(\langle a, j\rangle)$;
begin
if $\langle a, x.j\rangle \in A_{affected}$ **then**
Let $(\langle a, j\rangle(y_1, \ldots, y_k) = \zeta_{\langle a,j\rangle}) \in R(prod_{t'}(x))$ with $sort(\langle a, j\rangle) = \tau$;
if $x.j \in N_{new}$ **then** (* $\langle a, x.j\rangle \in A_{new}$ *)
$graph^{\tau}_{t'}(\langle a, x.j\rangle) := graph^{\tau}([\zeta_{\langle a,j\rangle}]_x)$;
foreach $\langle a', j'\rangle \in sym(\zeta_{\langle a,j\rangle}) \cap A(prod_{t'}(x))$ with $sort(\langle a', j'\rangle) = \tau'$ **do**
$graph^{\tau}_{t'}(\langle a, x.j\rangle) := graph^{\tau}_{t'}(\langle a, x.j\rangle)$
$[\langle a', x.j'\rangle \Leftarrow graph^{\tau'}(tree_{\Phi}(graph^{\tau'}_{t'}(\langle a', x.j'\rangle)))]$
end
elsif $x = x'$ **and** $j = 0$ **then** (* $\langle a, x.j\rangle \in A_{changed}$, but synthesized at x' *)
$OldValue := graph^{\tau}_{t'}(\langle a, x.j\rangle)$;
$graph^{\tau}_{t'}(\langle a, x.j\rangle) := graph^{\tau}([\zeta_{\langle a,j\rangle}]_x)$;
foreach $\langle a', j'\rangle \in sym(\zeta_{\langle a,j\rangle}) \cap A(prod_{t'}(x))$ with $sort(\langle a', j'\rangle) = \tau'$ **do**
$graph^{\tau}_{t'}(\langle a, x.j\rangle) := graph^{\tau}_{t'}(\langle a, x.j\rangle)$
$[\langle a', x.j'\rangle \Leftarrow graph^{\tau'}(tree_{\Phi}(graph^{\tau'}_{t'}(\langle a', x.j'\rangle)))]$
end
if $tree_{\Phi}(graph^{\tau}_{t'}(\langle a, x.j\rangle)) \neq tree_{\Phi}(OldValue)$ **then**
$AddDependentInstances(\langle a, j\rangle)$
else
$A_{affected} := A_{affected} \setminus \{\langle a, x.j\rangle\}$
end
else (* $\langle a, x.j\rangle \in A_{changed}$, somewhere else in the tree or inherited at x' *)
$OldValue := graph^{\tau}_{t'}(\langle a, x.j\rangle)$;
foreach $\langle a', j'\rangle \in sym(\zeta_{\langle a,j\rangle}) \cap A(prod_{t'}(x))$ with $sort(\langle a', j'\rangle) = \tau'$ **do**
if $\langle a', x.j'\rangle \in A_{affected}$ **then**
$graph^{\tau}_{t'}(\langle a, x.j\rangle) := graph^{\tau}_{t'}(\langle a, x.j\rangle)$
$[\langle a', x.j'\rangle \Leftarrow graph^{\tau'}(tree_{\Phi}(graph^{\tau'}_{t'}(\langle a', x.j'\rangle)))]$
end
end
if $tree_{\Phi}(graph^{\tau}_{t'}(\langle a, x.j\rangle)) \neq tree_{\Phi}(OldValue)$ **then**
$AddDependentInstances(\langle a, j\rangle)$
else
$A_{affected} := A_{affected} \setminus \{\langle a, x.j\rangle\}$
end
end
end
$PlanIndex := PlanIndex + 1$
end

Figure 27: *DoEval* procedure of the optimized incremental evaluator.

completely different structures. Nevertheless, the values which are represented by both graphs have to be checked for equality.

The computation of the graph of an instance $\langle a, x.j \rangle$ which is synthesized at x' is realized in the same way as new instances are evaluated (lines 13, ..., 17), after the old graph is stored in *OldValue* (line 12). After the computation the comparison of the graphs is done according to the *terms* they represent (line 18). Clearly, if the represented terms differ, the instances which depend on $\langle a, x.j \rangle$ may have changed, too. That's why they are added to the affected area (line 19) as it is done in the coarse-grain evaluator.

But the idea of *removing* the attribute instance $\langle a, x.j \rangle$ from the affected area (lines 21 and 34) is new. The benefit of this removal is that the set $A_{affected}$ is used not only for the detection of the instances which have to be recomputed but also for the detection of the extended edges which have to be replaced in order to obtain the new graph of a recomputed attribute instance.

Note that this technique would have already been possible for the coarse-grain evaluator, but it would not yield an optimization, since the currently treated attribute instance is never asked again, whether it is an element of the affected area!

Example 5.15 (Continuation of Example 5.13). To compare the optimized incremental evaluator with the incremental evaluator of Algorithm 4.16, we use the modification of the derivation tree t_{10}, which is illustrated in Figure 16.

The main steps done by the evaluator, i.e., updating the affected area and the calculation of a graph, are illustrated in Figure 28. ■

Comparing the incremental evaluator for the graph decoration with the incremental evaluator for the term decoration according to their runtime requirements, we observe two reasons to prefer the evaluator for the graph decoration.

Firstly, the graph representation allows to select only those parts of the graphs, which are affected by the subtree substitution. Thus, we avoid unnecessary reconstruction of the unchanged parts. The incremental evaluator for the term decoration does not have this facility. It has to fully build up the term of an attribute instance, if the attribute instance is detected to depend on an affected instance.

Secondly, the removal of attribute instances from the affected area reduces the number of extended edges which have to replaced in the recomputation of an attribute instance.

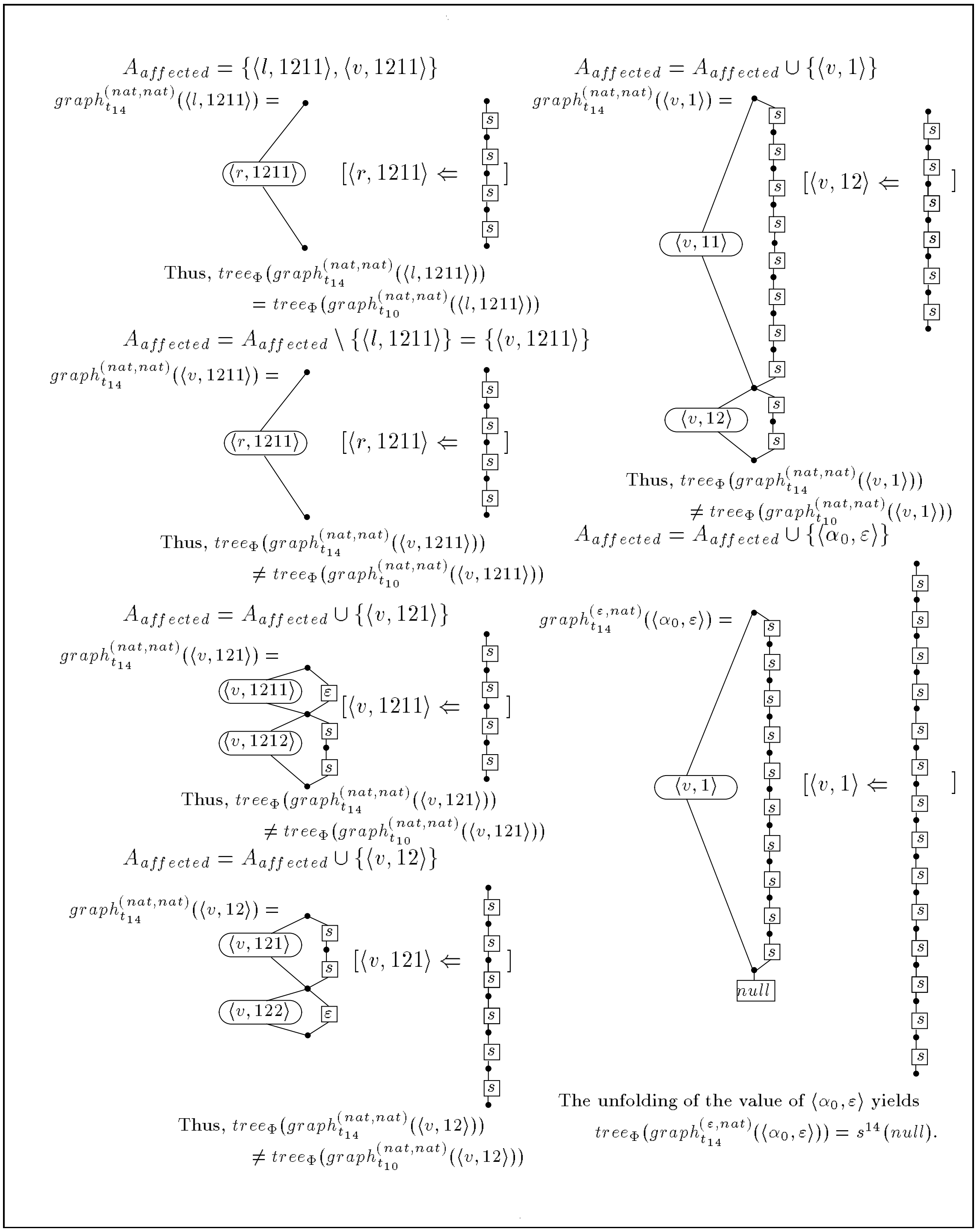


Figure 28: Evaluation steps to obtain the graph decoration of t_{14}.

References

[AM91] H. Alblas and B. Melichar. *International Summer School on Attribute Grammars, Applications, and Systems (Prague, Czechoslovakia)*, volume 545 of *LNCS*. Springer–Verlag, June 1991.

[CF82] B. Courcelle and P. Franchi–Zannettacci. Attribute grammars and recursive program schemes. *Theoret. Comput. Sci.*, 17:163–191, 235–257, 1982.

[Cou84] B. Courcelle. Attribute grammars: definitions, analysis of dependencies, proof methods. In B. Lorho, editor, *Methods and tools for compiler construction*, pages 81–102. Cambridge University Press, 1984.

[DJL86] P. Deransart, M. Jourdan, and B. Lorho. A survey on attribute grammars. Technical report, INRIA, 1986.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute grammars, definitions and bibliography*. Number 323 in Lect. Notes Comput. Sci. Springer–Verlag, 1988.

[DRT83] A. Demers, T. Reps, and T. Teitelbaum. Incremental context-dependent analysis for language-based editors. *ACM Trans. Programming Languages Systems*, 5:449–477, 1983.

[EF82] J. Engelfriet and G. File. Simple multi–visit attribute grammars. *J. Comput. System Sci.*, 24:283–314, 1982.

[EF89] J. Engelfriet and G. File. Passes, sweeps, and visits in attribute grammars. *J. ACM*, 36:841–869, 1989.

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal language theory: perspectives and open problems*, pages 241–286. New York, Academic Press, 1980.

[EV94] J. Engelfriet and H. Vogler. The translation power of top–down tree–to–graph transducers. *J. Comput. System Sci.*, 49:258–305, 1994.

[FHVV93] Z. Fülöp, F. Herrmann, S. Vágvölgyi, and H. Vogler. Tree transducers with external functions. *Theoret. Comput. Sci.*, 108:185–236, 1993.

[Fie93] J. Field. A graph reduction approach to incremental rewriting. *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, LNCS(690):259–273, 1993.

[Fül81] Z. Fülöp. On attributed tree transducers. *Acta Cybernet.*, 5:261–279, 1981.

[FV98] Z. Fülöp and H. Vogler. *Syntax-directed semantics — Formal Models Based on Tree Transducers*. Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1998.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN'84, Symposium on Compiler Construction, SIGPLAN Notices Vol.19, No.6*, pages 157–170, June 1984.

[Gie88] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Inform.*, 25:355–423, 1988.

[Hab92] A. Habel. *Hyperedge–replacement: Grammars and Languages*, volume 643 of *Lect. Notes Comput. Sci.* Springer–Verlag, 1992.

[HT86] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. SIGPLAN '86 Symposium on Compiler Construction, June 1986.

[Jok96] B. Jokubeit. Konstruktion eines inkrementellen Auswerters für geordnete Makroattributgrammatiken. Master's thesis, Dresden University of Technology, December 1996.

[Kas80] U. Kastens. Ordered Attribute Grammars. *Acta Inform.*, 13:229–256, 1980. Revised version in: Proc. of the Massey Functional Programming Workshop, ed. E. Ireland and N. Perry, 1992.

[Knu68] D.E. Knuth. Semantics of context–free languages. *Math. Systems Theory*, 2:127–145, 1968.

[KSV89] M.F. Kuiper, S.D. Swierstra, and H.H. Vogt. Higher order attribute grammars. In *Proc. SIGPLAN '89*, pages 131–145. ACM Press, 1989. Symp. Language Design and Implementation.

[Küh97] A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata.* PhD thesis, Dresden University of Technology, 1997.

[KV94] A. Kühnemann and H. Vogler. Synthesized and inherited functions — a new computational model for syntax–directed semantics. *Acta Inform.*, 31:431–477, 1994.

[KV97] A. Kühnemann and H. Vogler. *Attributgrammatiken.* Vieweg-Verlag, 1997.

[KW76] K. Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Conf. Rec. of 3rd Symp. on Principles of Programming Languages*, pages 32–49, 1976.

[LRS74] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed translations. *J. Comput. System Sci.*, 9:279–307, 1974.

[Mad97] W. H. Maddox. *Incremental Static Semantic Analysis.* Advanced research projects agency grant, USA, 1997.

[Paa95] J. Paakki. Attribute grammar paradigms — a high level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[RMT86] T. Reps, C. Marceau, and T. Teitelbaum. Remote attribute updating for language-based editors. *In Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 1–13, January 1986.

[RR93] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 502–510, 1993.

[RT89] T. W. Reps and T. Teitelbaum. *The synthesizer generator — a system for constructing language-based editors.* Springer–Verlag, 1989.

[vdM94] E. van der Meulen. *Incremental Rewriting.* PhD thesis, University of Amsterdam, 1994.

[WG84] W. M. Waite and G. Goos. *Compiler Construction.* Springer, 1984.

[Yeh83] D. Yeh. On incremental evaluation of ordered attributed grammars. *BIT 23*, pages 308–320, 1983.