CLASSICAL AND INCREMENTAL ATTRIBUTE EVALUATION
BY MEANS OF RECURSIVE PROCEDURES*

G. FILÉ
C.N.R.S., UER de Mathématiques et Informatique, Université de Bordeaux I, 33405 Talence Cedex, France

Abstract. The class of absolutely noncircular (ANC) attribute grammars (AGs) has been heavily studied, mainly because simple and recursive evaluators can be automatically produced for such grammars. We give a characterization of ANC AGs that includes as special cases most of the already existing definitions of this class. Our goal is that of clarifying the relationships among these definitions and also among the evaluators corresponding to them.

We show also that for a more restricted class of AGs (the doubly noncircular AGs) recursive incremental evaluators can be constructed in a way very similar to that used for the ANC AGs.

Introduction

The study of attribute grammars (AGs) has mainly concentrated on the problem of efficiently performing attribute evaluation. One class of AGs is particularly interesting in this respect: the class of the absolutely noncircular AGs (ANC AGs) [10]. This class is interesting because simple evaluators can be automatically constructed for them and because the problem of testing whether any given AG is ANC is only polynomial. For these reasons ANC AGs have been used in a compiler writing system [8] and studied in many theoretical papers [3, 4, 7, 9, 10, 13].

Several characterizations of the class of ANC AGs exist. We consider the three characterizations given by Kennedy and Warren [10], Courcelle and Franchi-Zanettacci [3, 4], and Jourdan [9], by Gombas and Bartha [7], and by Riis Nielson [13]. After some study, one realizes that all three characterizations can be viewed as requiring for an ANC AG the existence of some special graphs:

1. The original definition of Kennedy and Warren [10] states that an AG is ANC if, for each nonterminal X, there is a graph $D(X)$ with certain properties (given in Section 2). The nodes of $D(X)$ are the attributes of X and the arcs run only from inherited to synthesized attributes (abbreviated as i- and s-attributes in what follows). Intuitively, $D(X)$ gives for each s-attribute $a_0$ of X the i-attributes $a_1, \ldots, a_k$ of X that are always sufficient for computing $a_0$; $a_1, \ldots, a_k$ are all the i-attributes that are sources of arcs of $D(X)$ entering $a_0$; they are said to be connected to $a_0$.

* The work reported in this article was carried out with the support of the group ATTRISEM of the GRECO de programmation.
(2) The characterization given by Gombas and Bartha [7] can be viewed as generalizing the original definition in the following sense: an AG $G$ is ANC iff, for each nonterminal $X$ of $G$, there exists a graph $S(X)$ satisfying properties similar to those of $D(X)$ above, but different from $D(X)$ in the fact that the nodes of $S(X)$ are now sets of $i$- or of $s$-attributes of $X$ (and no longer singletons as for $D(X)$). The arcs of $S(X)$ still run only from sets of $i$-attributes to sets of $s$-attributes.

(3) Finally, it is shown by Riis Nielson [13] that an AG $G$ is ANC iff, for each nonterminal $X$ of $G$, there exists a set $T(X)$ of noncircular graphs, satisfying properties similar to those of the graphs $D(X)$ and $S(X)$, but which, moreover, define a total order of their nodes, i.e., for any two nodes of such graphs, there must be a path from one to the other. Such graphs are called totally ordered. For satisfying this property, the graphs in $T(X)$ are allowed to contain arcs running from sets of $s$-attributes to sets of $i$-attributes, in addition to the arcs running in the opposite direction of the graphs $S(X)$ and $D(X)$.

An interesting aspect of these graph-based characterizations of ANC AGs is that they lend themselves naturally to the definition of recursive evaluators for these grammars.

Recursive evaluators for ANC AGs are described already in [10], but they are quite complicated. In [3, 4, 9] it is shown how very simple evaluators can be constructed from the graphs $D(X)$ of [10]. These evaluators consist of a set of recursive procedures each of which has the task of computing one $s$-attribute $a_0$ of a nonterminal node, say $X$, in all occurrences of $X$ in the input tree. Intuitively, one can think that such a procedure has as parameters the values of the $i$-attributes $a_1, \ldots, a_k$ connected to $a_0$ in $D(X)$.

This evaluator model suffers from two drawbacks:

(i) It computes one attribute at the time (one procedure for each $s$-attribute) and, thus, it does not try to optimize the evaluation by 'computing attributes together'.

(ii) It may compute some attributes of an input tree more than once. Intuitively, this is due to the fact that the procedures for computing different attributes ignore each other and, thus, for any two such procedures, the second one may recompute attributes already computed by the first.

The evaluators proposed in [7], based on the characterization (2) above, do not suffer of drawback (i). For each block $B$ of $s$-attributes in $S(X)$ the evaluator has one recursive routine for computing the attributes in $B$ for every occurrence of $X$ in the input tree. But, clearly, different procedures are still independent and, thus, drawback (ii) is still present in these evaluators.

The evaluators proposed in [13] (thus based on the characterization (3)) solve both drawbacks. The reason why they do not recompute is that the totally ordered graphs of $T(X)$ allow ordering the different recursive procedures in the following sense: a procedure $P$ for computing a block $B$ of $s$-attributes of a node $n$, labeled by, say $X$, 'arrives' in $n$ with a graph $g \in T(X)$, and $g$ 'tells' to $P$ which procedures have already been called at $n$; in this way the procedure for $B$ 'knows' what is already computed and avoids to do it again. Intuitively, a path in $g$ from a block
Attribute evaluation by recursive procedures

B of s-attributes to another block \(B'\) of s-attributes implies that the procedure for computing \(B\) is called before that for computing \(B'\).

Unfortunately, these evaluators have a new drawback. Their size depends on that of the sets of graphs \(T(X)\) (whereas the preceding evaluators need only one graph for a nonterminal node) and \(T(X)\) may contain exponentially many graphs in the number of attributes of \(X\). Thus we are confronted with a classical case of trade-off: to solve drawback (ii) we risk having to build very large evaluators.

In this paper we further develop the unifying graph-based view, described above, for three purposes:

(a) to clarify the relation among most of the already existing results and definitions concerning ANC AGs and also the relation between each of these definitions and the efficiency of the corresponding recursive evaluators;

(b) to show that another class of AGs, called doubly noncircular [1], can be characterized by means of graphs in a way very similar to that of ANC AGs, and that these graphs are a natural basis for the construction of incremental recursive evaluators for the doubly noncircular AGs;

(c) to show clearly how both the classical evaluators for the ANC AGs and the incremental ones for the doubly noncircular AGs can be constructed automatically.

We show that an AG \(G\) is ANC iff, for each nonterminal \(X\), there is a set \(\Pi(X)\) of noncircular graphs that satisfies again the properties of characterizations (1)-(3), but for which no assumption is made about the arcs. These graphs are called partially ordered. Clearly, our \(\Pi A(X)\) occupies an intermediate position between characterizations (2) and (3), in the sense that from \(\Pi A(X)\) one can obtain a set \(T(X)\) as in (3) and a graph \(S(X)\) as in (2), roughly, by respectively adding arcs to \(\Pi A(X)\) and eliminating arcs from \(\Pi A(X)\).

In the same way, the evaluators corresponding to the graphs \(\Pi A(X)\) occupy an intermediate position (w.r.t. the size and the recomputation) between the evaluators based on the graphs \(S(X)\) and those based on the sets of graphs \(T(X)\). This is interesting because by 'tuning' carefully the graphs \(\Pi A(X)\) one may hope to construct evaluators that realize a good compromise between their size and their amount of recomputation.

Whereas for ANC AGs we essentially present a unifying frame for better comparing already known results, the application of these ideas to the doubly noncircular AGs is new. This class of AGs was introduced in [1] under the name of 'ordre partiel clos' AGs: for uniformity, we prefer to call it doubly noncircular (abbreviated as DNC) AGs.

We show that recursive incremental evaluators can be constructed for DNC AGs. These evaluators are interesting mainly for two reasons:

(i) They are a natural extension of the classical recursive evaluators for ANC AGs: in addition to the recursive procedures for computing sets of s-attributes, which 'descend' the input tree, they have procedures for computing sets of i-attributes, which use the recursion for 'climbing up' the tree.

(ii) As a consequence of (i), they can be constructed in a way very close to that
of the classical evaluators of the ANC AGs and, thus, their construction can be easily automated.

For stressing point (ii), our description of the incremental evaluators is quite detailed, see Construction 3 in Section 4. We also prove the correctness and the optimality of our construction.

The remainder of the paper consists of five parts. Section 1 contains the usual definitions of AGs and related concepts. Section 2 contains the definitions of the new notions and the unifying characterizations of ANC and DNC AGs discussed above. In Section 3 the construction of the classical evaluators for the ANC AGs is described and proved correct. Section 4 contains the construction of the incremental evaluators for the DNC AGs and the proof of its correctness and optimality. The paper ends with a short conclusion.

1. Preliminaries

Let us start by briefly recalling the definition of attribute grammars [11]. A more formal definition can be found in [5].

Definition 1.1. An attribute grammar (AG) is a 4-tuple \( G = (G_0, \text{ATT}, V, \text{SEM}) \).

(1) \( G_0 \) is a context-free grammar \((N, T, P, Z)\), called the underlying context-free grammar of \( G \), where, as usual, \( N \) is the set of the nonterminal symbols, \( T \) that of terminal symbols, \( P \) that of productions, and \( Z \) is the initial symbol of \( G \). \( Z \) appears only at the left-hand side of productions. A production \( p \in P \) has the form \( X_0 \rightarrow w_0X_1 w_1 \ldots w_{\gamma-1}X_{\gamma} w_{\gamma} \), where \( \gamma \geq 0 \), \( X_i \in N \), and \( w_i \in T^* \) for \( i \in [0, \gamma] \). Since the terminals are not important for our study, we will very often drop them and represent \( p \) simply as \( X_0 \rightarrow X_1 \ldots X_{\gamma} \). We assume that the derivation trees of \( G_0 \) have all leaves labeled by terminal symbols; such a tree is complete if its root is \( Z \).

(2) \( \text{ATT} \) is a function, \( N \rightarrow 2^A \), where \( A \) is a set of symbols called attribute names, or simply attributes. The set \( A \) is partitioned into two subsets: that of the inherited and that of the synthesized attributes, shortened as i- and s-attributes, respectively. For \( X \) in \( N \), \( \text{ATT}(X) \) is the set of attributes of \( X \). \( \text{IATT}(X) \) and \( \text{SATT}(X) \) are the sets of the i- and s-attributes of \( X \). The assumption is made that \( \text{IATT}(Z) = \{d\} \) and that \( \text{SATT}(Z) \) contains an attribute, denoted \( d \), designated to always hold the translation of any complete derivation tree of \( G \) (this notion will be explained later).

(3) \( V: A \rightarrow \text{Domains} \), gives for each attribute \( b \) its domain of values \( V(b) \).

(4) For each production \( p \in P \), \( \text{SEM}(p) \) is the set of the semantic rules of \( p \). This set is defined as follows: Let \( p \) be \( X_0 \rightarrow X_1 \ldots X_{\gamma} \); an attribute of \( p \) is a pair \((a, j)\), where \( j \in [0, \gamma] \) and \( a \in \text{ATT}(X_j) \). A semantic rule \( \alpha \) in \( \text{SEM}(p) \) has the form

\[
(a_0, i(0)) = f((a_1, i(1)), \ldots, (a_m, i(m))),
\]

where, for \( j \in [0, m] \), \( i(j) \in [0, \gamma] \), and \( f \) is a strict mapping of type, \( V(a_1) \times \cdots \times V(a_m) \rightarrow V(a_0) \). Because of such a semantic rule \( \alpha \), one says that \((a_0, i(0))\) is defined
using \((a_1, i(1)), \ldots, (a_m, i(m))\) as parameters, and also that \((a_0, i(0))\) depends on these parameters in \(p\).

Let \(\text{Def}(p)\) be the set of the attributes \((a, j)\) of \(p\) such that, either \(j = 0\) and \(a\) is synthesized, or \(j \geq 1\) and \(a\) is inherited. \(\text{App}(p)\) contains the remaining attributes of \(p\). (\(\text{Def}\) and \(\text{App}\) stand for defined and applied, respectively, the reason for this should become clear immediately below.)

The following usual assumptions are made about the set \(\text{SEM}(p)\):

(i) There must be a bijection \(\delta: \text{Def}(p) \to \text{SEM}(p)\), such that if \(\alpha = \delta((a, j))\), then \((a, j)\) is the left-hand side of \(\alpha\).

(ii) The attributes that are parameters in the semantic rules of \(\text{SEM}(p)\) must all be in \(\text{App}(p)\) (see [2]).

We will mainly consider AGs from a ‘schematic viewpoint’, that is, we will not be interested in what the AGs compute, but only in the dependency relation that the semantic rules determine among the attributes. This relation is interesting because it reflects the order in which the attributes must be computed.

It is useful (and classical in this domain) to represent the dependency relation by means of graphs. The dependencies among the attributes of a production \(p\) are represented by the dependency graph of \(p\), denoted \(D(p)\), as follows: the nodes of \(D(p)\) are all the attributes of \(p\), and \(D(p)\) has an arc from the used attribute \((b, i)\) to the defined attribute \((a, j)\) iff in \(p\) \((a, j)\) is defined using \((b, i)\) as a parameter. In Fig. 1 the dependency graphs of the productions of the AG GEX are presented. GEX will be our running example throughout the paper. In Fig. 1, for \(j \in [1, 3]\), \(i_j\) is an \(i\)-attribute of \(X\) and \(s_j\) an \(s\)-attribute of \(X\); \(d\) is the designated \(s\)-attribute of \(Z\).

For representing derivation trees, we will use the Dewey notation with the following conventions: the root of a tree is denoted by 0, the sons of a node \(n\) are \(n_1, \ldots, n_k\), and \(n_0\) represents the same node as \(n\). \(\text{lab}(n)\) is the label of node \(n\). An occurrence of a production \(p: X_0 \Rightarrow X_1 \ldots X_r\) in a derivation tree is denoted by \(\langle n_0, n_1, \ldots, n_r \rangle\), where \(p\) is applied to \(n_0\). For a derivation tree \(t\), an attribute of a node \(n\) of \(t\) (and of \(t\) itself) is a couple \((a, n)\), where \(a \in \text{ATT}(\text{lab}(n))\).

As for productions we define the dependency graph of a derivation tree \(t\), denoted \(D(t)\), as follows: \(D(t)\) has all attributes of \(t\) as nodes, and, for every occurrence \(\langle n_0, \ldots, n_\gamma \rangle\) of a production \(p\) in \(t\), \(D(t)\) contains an arc from \((a, n_j)\) to \((b, n_i)\), \(j\) and \(i\) in \([0, \gamma]\), iff \(D(p)\) contains an arc from \((a, j)\) to \((b, i)\).

Given an AG \(G = (G_0, \text{ATT}, V, \text{SEM})\), an evaluator for \(G\) is any program that computes for every complete derivation tree \(t\) of \(G\) a consistent valuation for \(t\).

For any derivation tree \(t\), a valuation of \(t\) is a function

\[
\text{VAL} : \{\text{attributes of } t\} \to \bigcup \{V(a) \mid a \in A\} \cup \{\bot\}.
\]

An attribute \((a, n)\) is undefined in \(\text{VAL}\) if \(\text{VAL}(a, n) = \bot\), otherwise it is defined and \(\text{VAL}(a, n)\) is its value in \(\text{VAL}\). A valuation \(\text{VAL}\) for \(t\) is consistent if for each attribute \((a, n)\) such that \(\text{VAL}(a, n) \neq \bot\), either (i) or (ii) below holds:

(i) If \(a \in \text{SATT}(\text{lab}(n))\), then, if \(p\) is the production applied to \(n\), \(\text{SEM}(p)\) contains
a semantic rule defining \((a, 0)\), say

\[
(a, 0) = f((a_1, i(1)), \ldots, (a_m, i(m))),
\]

then it must be that

\[
\text{VAL}(a, n) = f(\text{VAL}(a_1, ni(1)), \ldots, \text{VAL}(a_m, ni(m))).
\]

(ii) If \(a \in \text{IATT}(\text{lab}(n))\), then a condition similar to (i) must hold, the only difference being that the semantic rule defining \((a, n)\) belongs to the production applied at the father of \(n\).

Observe that since the functions used in the semantic rules are assumed to be strict, for a consistent valuation \(\text{VAL}\), if an attribute is defined, then all the attributes it depends on must also be defined in \(\text{VAL}\).

For a valuation \(\text{VAL}\) of a tree \(t\), the attributes whose values do not satisfy the conditions (i) and (ii), are called inconsistent. If all attributes of \(t\) are defined in \(\text{VAL}\), then \(\text{VAL}\) is said to be full and \(t\) (together with \(\text{VAL}\)) is said to be fully attributed.

If \(\text{VAL}\) is a consistent valuation of \(t\), if the root of \(t\) is labeled by \(Z\) and \(\text{VAL}(d, 0)\) is defined in \(\text{VAL}\), then this value is the translation of \(t\).

Since in what follows we will often handle graphs, the following notation will be of use: for a directed graph \(g = (V, E)\), where \(V\) is the set of nodes and \(E\) that of arcs, for any node \(m \in V\), the subgraph of \(g\) connected to \(m\) is the graph \(g' = (V', E')\),
where \( V' \subseteq V \) is such that \( v \in V' \) iff \((v, m) \in E' \) and \( E' = E \cap V'^2 \); a sequence \( \langle n_1, \ldots, n_k \rangle \) of some nodes of \( g \) is said to respect \( g \) if for no two \( i \) and \( j \) such that \( i \) and \( j \) are in \([1, k]\) and \( i < j \), there is a path in \( g \) from \( n_i \) to \( n_j \).

2. The characterization of ANC and DNC attribute grammars

In this section several new concepts are introduced and illustrated by means of examples. At the end of the section these notions are used for giving the unifying characterization of ANC and DNC AGs that was announced in the Introduction.

Definition 2.1. A partially ordered partition (po-partition) of a set \( S \) is a couple \( \Pi(S) = (M, \rightarrow) \), where \( M \) is a partition of \( S \) and \( \rightarrow \) is a relation on \( M \). \( \Pi(S)^+ \) is \( \langle M, \rightarrow^+ \rangle \). If \( \rightarrow^+ \) defines a total order on \( M \), then \( \Pi(S) \) is a totally ordered partition of \( S \) (to-partition). Clearly, \( \Pi(S) \) can be viewed as a graph on \( M \) and this point of view will be often used in the sequel.

Definition 2.2. For an AG \( G \), an assignment of po-partitions for \( G \) (shortly, a pop-assignment for \( G \)) is a family of sets \( \Pi A = \{ \Pi A(X) \mid X \) is a nonterminal node of \( G \} \), where \( \Pi A(X) = \{ \Pi_\ell, \ldots, \Pi_k \} \) and each \( \Pi_\ell \) is a po-partition of \( \text{ATT}(X) \) such that if \( \Pi_\ell = (M, \rightarrow) \), each element of \( M \) contains either only i-attributes or only s-attributes (or it is empty). If for every nonterminal \( X \) every \( \Pi \in \Pi A(X) \) is a to-partition, then \( \Pi A \) is a top-assignment for \( G \).

There are three restricted types of pop-assignments that are useful to consider:

1. a pop-assignment \( \Pi A \) is trivial if, for every nonterminal \( X \), \( \Pi A(X) = \{ \Pi \} \) and \( \Pi = (M, \rightarrow) \) such that \( M = \{ \{ a \} \mid a \in \text{ATT}(X) \} \);

2. a pop-assignment \( \Pi A \) is IS if, for every nonterminal \( X \) and every \( (M, \rightarrow) \in \Pi A(X) \), \( \rightarrow \) runs only from blocks of i-attributes to blocks of s-attributes;

3. a pop-assignment \( \Pi A \) is ISI if, for any \((M, \rightarrow)\) as above, \( \rightarrow \) runs either from blocks of i-attributes to blocks of s-attributes or vice versa.

Example 2.3 (pop-assignments for \( GEX \)). Two pop-assignments \( \Pi A' \) and \( \Pi A \) for the AG \( GEX \) of Fig. 1 are given below:

1. \( \Pi A' \) is defined as follows:
   
   (a) \( \Pi A'(X) = \{ \Pi_\ell \} \), where \( \Pi_\ell = (\{ d \}, \emptyset) \);

   (b) \( \Pi A'(X) = \{ \Pi_1', \Pi_2' \} \), where, using the graph representation of po-partitions,

   \[
   \Pi_1': \{ i_1 \} \rightarrow \{ i_2, i_3 \}, \{ s_2 \} \rightarrow \{ s_1, s_3 \},
   
   \Pi_2': \{ i_1, i_2, i_3 \} \rightarrow \{ s_1, s_2, s_3 \};
   \]

   \]
(c) $IIA'(Y) = \{II_Y\}$, where

$$II_Y: \{i\} \rightarrow \{s\}.$$

(2) $IIA$ is defined as follows:

(a) $IIA(Z) = \{II_Z\}$ and $IIA(Y) = \{II_Y\}$;

(b) $IIA(X) = \{II_1, II_2\}$, where

$$II_1: \{i_1\} \rightarrow \{s_1\} \quad \{i_2\} \rightarrow \{s_2\} \quad \{i_3\} \rightarrow \{s_3\},$$

$$II_2: \{i_2\} \rightarrow \{s_2\} \quad \{i_1, i_3\} \rightarrow \{s_1, s_3\}.$$

Note that $IIA$ is an IS pop-assignment.

We want to use po-partitions for constructing new graphs from the dependency graphs of the productions of an AG.

**Definition 2.4.** Consider an AG $G$, a production $p: X_0 \rightarrow X_1 \ldots X_{\gamma}$ of $G$ and a sequence of po-partitions $II_0 \in IIA(X_0), \ldots, II_{\gamma} \in IIA(X_{\gamma})$, respectively. $D(p)[II_0, \ldots, II_{\gamma}]$ is the graph defined as follows:

(i) the nodes are all the couples $(A, i)$, where $A \in M_i$, $i \in [0, \gamma]$,

(ii) there are two types of arcs:

(a) arcs that come from each $II_i$: for each $i \in [0, \gamma]$, if $A \rightarrow B$, $A$ and $B$ in $M_i$, then there is an edge running from $(A, i)$ to $(B, i)$,

(b) arcs that come from $D(p)$: if attribute $(a, i)$ of $p$ depends in $p$ on attribute $(b, j)$ and $a \in A \in M_i$ and $b \in B \in M_j$, then there is an edge running from $(B, j)$ to $(A, i)$.

The graph contains no other arc.

With $D(p - j)[II_0, \ldots, II_{\gamma}]$, $j \in [0, \gamma]$, we denote the graph obtained from $D(p)[II_0, \ldots, II_{\gamma}]$ by eliminating those edges of (a) coming from $II_j$.

**Definition 2.5.** Given a graph $g = D(p)[II_0, \ldots, II_{\gamma}]$ as described in Definition 2.4, for $i \in [0, \gamma]$, $\Phi_ig$ is the projection of the transitive closure of $D(p - i)[II_0, \ldots, II_{\gamma}]$ into the nodes $\{(A, i) | A \in M_i\}$.

**Example 2.6.** Figure 2 shows the graph $g_1 = D(1)[II_Z, II_Y, II_{\gamma}]$, where production 1 is shown in Fig. 1 and the po-partitions $II_Z, II_Y$, and $II_{\gamma}$ are defined in point (1) of Example 1. Figure 2 shows also the graphs $\Phi_1g_1$ and $\Phi_2g_1$.

Figures 3, 4, and 5 represent, respectively, the graphs $g_2 = D(1)[II_Z, II_Y, II_{\gamma}]$, $g_3 = D(2)[II_1, II_Y, II_{\gamma}]$, and $g_4 = D(2)[II_Y, II_Y, II_{\gamma}]$. It is natural to extend the notion of attribute of a production to that of block of attributes of a production. Hence, the block of attributes $\{i_1, i_3\}$ on the left-hand side of the production in Fig. 5 will be denoted by $\{(i_1, i_3), 0\}$, whereas that of the $X$ on the right-hand side of the production is $\{(i_1, i_3), 2\}$.

**Definition 2.7.** For an AG $G$, a pop-assignment $IIA$ for $G$ is good if for any production $p: X_0 \rightarrow X_1 \ldots X_{\gamma}$ of $G$ and po-partition $II_0 \in IIA(X_0)$ the following
holds: there exists a choice $\Pi_1, \ldots, \Pi_r$, in $IIA(X_1), \ldots, IIA(X_r)$, respectively, such that,

(i) $D(p)[\Pi_0, \ldots, \Pi_r]$ is noncircular, and

(ii) $\Phi_0 D(p)[\Pi_0, \ldots, \Pi_r]$ is a subgraph of $\Pi_0$.

**Example 2.8 (good pop-assignment).** For our example $AG$ $GEX$ the pop-assignment $IIA$ of Example 2.3 is good: in Fig. 3 $g_2$ shows that $IIA$ satisfies the conditions of Definition 2.7 for production 1 and $g_4$ of Fig. 5 shows that these conditions are
satisfied for production 2 when $\Pi_2$ is taken for the left-hand side. It is easy to see that Definition 2.7 is satisfied in all other cases. At the contrary, the pop-assignment $\Pi A'$ given in Example 2.3 is not good for GEX: consider $g = D(3)[\Pi_1']$; $\Phi_0g$ is not a subgraph of $\Pi_1'$.

**Definition 2.9.** For an AG $G$, a pop-assignment $\Pi A$ for $G$ is fine if for every production $p : X_0 \rightarrow X_1 \ldots X_\gamma$ of $G$, for every $i \in [0, \gamma]$, and for every po-partition $\Pi_i \in \Pi A(X_i)$, there exists at least one choice of po-partitions: $\Pi_0 \in \Pi A(X_0), \ldots, \Pi_{i-1} \in \Pi A(X_{i-1}), \Pi_{i+1} \in \Pi A(X_{i+1}), \ldots, \Pi_\gamma \in \Pi A(X_\gamma)$, such that,

(i) $D(p)[\Pi_0, \ldots, \Pi_\gamma]$ is noncircular, and

(ii) $\Phi_iD(p)[\Pi_0, \ldots, \Pi_\gamma]$ is a subgraph of $\Pi_i$.

**Example 2.10 (fine pop-assignment).** The pop-assignment $\Pi A$ of Example 2.3 is
not fine for GEX: consider Fig. 3, $\Phi_1 g_2$ is not a subgraph of $\Pi_i$ (and similarly for $\Phi_2 g_2$ and $\Pi_i$). One could think that it would suffice to include the missing dependencies into $\Pi_i$ and $\Pi_i$, i.e., to consider $\Pi_i = \Pi_i \cup \Phi_1 g_2$ and $\Pi_i = \Pi_i \cup \Phi_2 g_2$. This is not true. Intuitively, the reason is the following: production 1, cf. Fig. 1, through the dependencies $(s_1, 1) \rightarrow (i_2, 2)$ and $(s_2, 2) \rightarrow (i_2, 1)$, ‘forces’ $(i_1, 1)$ and $(s_1, 1)$ to be before $(i_2, 1)$ and $(s_2, 1)$ and, through $(s_2, 2) \rightarrow (i_2, 1)$ and $(s_2, 1) \rightarrow (i_1, 2)$, it also ‘forces’ $(i_2, 2)$ and $(s_2, 2)$ to be before $(i_1, 2)$ and $(s_1, 2)$. Clearly, these dependencies, put together, would give a circular po-partition, hence one needs at least two po-partitions for $X$, say $\Pi_a$ and $\Pi_b$, $\Pi_a$ for the first occurrence of $X$ in production 1 and $\Pi_b$ for the second occurrence. But then it is not possible to satisfy condition (1) of Definition 2.9 in the case one chooses $\Pi_a$ for the second occurrence of $X$ in production 1, or, equivalently, $\Pi_b$ for the first occurrence.

One could formally prove that GEX does not admit any fine pop-assignment. However, it is easy to modify GEX in such a way that a fine pop-assignment exists for the new AG: it suffices to eliminate from production 1 the dependency $(s_2, 1) \rightarrow (i_1, 2)$. For obtaining a more interesting AG let us replace production 1 of GEX by the production shown in Fig. 6. GEX' is the AG one obtains in this way and what is shown in Fig. 6 is its production 1.

It is easy to see that the following pop-assignment $\overline{\Pi A}$ is fine for GEX':

$$\overline{\Pi A}(Z) = \{\Pi_Z\}, \quad \overline{\Pi A}(Y) = \{\Pi_Y\}, \quad \overline{\Pi A}(X) = \{\Pi_X\},$$

where

$$\Pi_X : \{i_1, i_3\} \rightarrow \{s_1, s_3\} \rightarrow \{i_2\} \rightarrow \{s_2\}.$$

Let us recall the definitions of the classes of absolutely noncircular and doubly noncircular AGs reformulating them in our terminology. These classes were defined originally in [10] (see also [5]) and [1], respectively. In [1] the class of doubly noncircular AGs was called OPC for ‘ordre partiel clos’.
Definition 2.11. (1) An AG $G$ is absolutely noncircular (ANC) if there exists a trivial IS pop-assignment that is good for $G$.
(2) An AG $G$ is doubly noncircular (DNC) if there exists a trivial ISI pop-assignment that is fine for $G$.

In the following theorem we show that Definition 2.11 can be generalized in several ways giving rise to characterizations of ANC and DNC AGs that are more useful for the automatic construction of efficient evaluators for them.

Theorem 2.12. Let $G$ be an AG.
(1) The following four points are equivalent:
   (a) $G$ is ANC,
   (b) there is an IS good pop-assignment $IIA$ for $G$ such that, for each nonterminal $X$ of $G$, $IIA(X)$ is a singleton [7],
   (c) there is a good pop-assignment for $G$,
   (d) there is a good top-assignment for $G$ (cf. Definition 2.2) [13].
(2) The following four points are equivalent:
   (a) $G$ is DNC,
   (b) there is an ISI fine pop-assignment $IIA$ for $G$ such that, for each nonterminal $X$ of $G$, $IIA(X)$ is a singleton,
   (c) there is a fine pop-assignment for $G$,
   (d) there is a fine top-assignment for $G$.

Proof. The proofs of (1) and (2) are similar. Therefore only that of (1) will be given. (a) $\Rightarrow$ (b) $\Rightarrow$ (c) is evident.

(b) $\Rightarrow$ (a) is also easy: Let $IIA(X) = \{II = (M, +)\}$, and $D(X)$ be the po-partition $\langle\{a\} | a \in ATT(X)\rangle$, where $\{a\} \rightarrow \{b\}$ iff $a \in A \in M, b \in B \in M$, and $A \rightarrow B$; $IIA'$ defined by $IIA'(X) = \{D(X)\}$ is a trivial IS good pop-assignment for $G$.

(c) $\Rightarrow$ (b) is shown in two steps:
Step 1. Transform any good pop-assignment $IIA$ for $G$ into an IS good pop-assignment $IIA'$ for $G$. Consider $II \in IIA(X)$ for any nonterminal $X$ and let $II'$ be the po-partition one obtains from $II$ by taking its transitive closure and then deleting all arcs that do not satisfy the IS property. One can readily show that $IIA'$, defined by $IIA'(X) = \{II' | II \in IIA(X)\}$, is good for $G$.

Step 2. Transform any IS good-pop-assignment $IIA$ for $G$ into one $IIA''$ as in (b). Let $IIA(X) = \{II_1, \ldots, II_k\}$, with $II_i = (M_i, \rightarrow_i, i \in [1, k]$. From $IIA(X)$ one constructs $IIA''(X) = (M'', \rightarrow'' )$ as follows:

(i) Let $P$ be the set of all nonempty subsets $A$ of $ATT(X)$ such that for each $j \in [1, k]$ there is an element of $M_j$ that contains $A$. $M''$ is the set of the maximal elements of $P$ w.r.t. the inclusion.

(ii) Let $M'' = \{A_1, \ldots, A_q\}$ and let $M_j(A_i), i \in [1, q]$ and $j \in [1, k]$, denote the element of $M_j$ that contains $A_i$; the relation $\rightarrow''$ is as follows: $A_i \rightarrow'' A_p$ iff for every $j \in [1, k], M_j(A_i) \rightarrow_j M_j(A_p)$. Observe that, since $IIA$ is IS, $IIA''(X)$ respects the IS
property. It is easy to see that IIA" defined by IIA"(X) = {II"(X)} for every nonterminal X, is good for G.

(d) ⇒ (c) is obvious.

For showing that (c) ⇒ (d) it suffices to observe the following (this argument is taken from [13]): Consider a good pop-assignment IIA for G, and, for any production

\[ p : X_0 \rightarrow X_1 \ldots X_\gamma \]

of G, consider a graph \( g = D(p)[\Pi_{i_0}, \ldots, \Pi_{i_\gamma}] \), with \( \Pi_i = \langle M_i, \rightarrow \rangle \in IIA(X_i), i \in [0, \gamma] \), such that g satisfies the two conditions of Definition 2.7. Let \( \Pi_0 \) be a to-partition \( \langle M_0, \rightarrow_0 \rangle \), where the order on the elements of \( M_0 \) defined by \( \rightarrow_0 \) respects \( \Pi_0 \). The graph \( g' = D(p)[\Pi_0', \Pi_1', \ldots, \Pi_{\gamma}'] \) is acyclic: otherwise \( \Phi g \) would not be a subgraph of \( \Pi_0 \). Since \( g' \) is acyclic, one can order its nodes in a way that respects it. Clearly, this order defines to-partitions, \( \Pi_0' = \langle M_1, \rightarrow_1 \rangle, \ldots, \Pi_{\gamma}' = \langle M_{\gamma}, \rightarrow_\gamma \rangle \), such that \( D(p)[\Pi_{i_0}', \ldots, \Pi_{i_\gamma}'] \) satisfies Definition 2.7.

The above observation shows that the following top-assignment IIA' is good for G: for each \( \Pi = \langle M, \rightarrow \rangle \in IIA(X) \), let \( BII \) be the set of the to-partitions \( \Pi' = \langle M, \rightarrow' \rangle \) such that \( \rightarrow' \) defines a total order on \( M \) that respects \( \Pi \); for each nonterminal \( X \), \( IIA'(X) = \bigcup \{ BII | \Pi \in IIA(X) \} \).

For concluding this section let us review some complexity results concerning the classes of AGs under consideration.

1. It is well known that one can decide whether an AG is ANC in time polynomial in its size [10]. A similar result can easily be shown for DNC AGs.

2. In [6] it is shown that the following problem is NP-complete: decide whether an AG allows a good top-assignment IIA such that, for each nonterminal \( X \), \( IIA(X) \) is a singleton. This class is called simple multi-visit and also \( l \)-ordered. It is not difficult to generalize the proof of [6] in order to show that for any \( k \geq 2 \) the following problem is also NP-complete: decide whether for any given AG there exists a good top-assignment such that for each \( X \), \( IIA(X) \) contains at most \( k \) to-partitions.

3. In [7] the proof from [6] is slightly modified for proving the NP-completeness of the following problem: decide whether an AG allows a good pop-assignment IIA such that for each \( X \), \( IIA(X) = \{ \langle M, \rightarrow \rangle \} \), where \( M \) contains at most two elements.

3. The classical evaluators

In this section we will show how, given an ANC AG \( G \) and a pop-assignment for it, one can construct an evaluator for \( G \). The correctness of the construction will also be proved.

3.1. Construction 1: The ANC evaluator

In what follows let \( G \) be an ANC AG and IIA a good pop-assignment for \( G \); for any production \( p : X_0 \rightarrow X_1 \ldots X_\gamma \) of \( G \) and \( \Pi_0 \in IIA(X_0) \), \( \Omega(p, \Pi_0) \) is a sequence
\(\langle \Pi_1, \ldots, \Pi_r \rangle\) of po-partitions in \(\Pi A(X_1), \ldots, \Pi A(X_r)\), respectively, such that \(D(p)[\Pi_0, \ldots, \Pi_r]\) satisfies both conditions of Definition 2.7. \(\Omega\) is called a pop-function for \(G\) and \(\Pi A\). For simplicity and without loss of generality, we assume that \(\Pi A\) is ISI and that \(\Pi A(Z) = \{\Pi Z\}\).

An ANC evaluator for \(G\) consists of a set of recursive procedures and of a main program.

3.1.1. The recursive procedures

For each nonterminal \(X\) of \(G\), po-partition \(\Pi_0 \in \Pi A(X)\), and block \(B\) of \(s\)-attributes of \(\Pi_0\), there is a procedure, called \(\Pi_0 B\), that has one formal parameter of type node of the input tree. The evaluator calls \(\text{eval } \Pi_0 B\) in order to compute the block of attributes \((B, n)\). Clearly, \(\text{lab}(n)\) must be \(X\). The structure of \(\text{eval } \Pi_0 B\) is shown in Fig. 7, where \(p_1, \ldots, p_m\) are all the productions of \(G\) with \(X\) at the left-hand side and \(\text{OPT}(p_1, B)\) is a sequence of instructions that is described in Fig. 7.

```
procedure eval \Pi_0 B(n);
begin
  case production applied at \(\Pi_0\) of \(p\) of \(\Pi_0 B\):
    \(p_1: \text{OPT}(p_1, B);\)
    \(\vdots\)
    \(p_m: \text{OPT}(p_m, B);\)
  end of case;
end;
```

For any input tree \(t\) the ANC evaluator uses a variable \(\text{value}(a, n)\) for each attribute \((a, n)\) of \(t\). Clearly, the evaluator stores in \(\text{value}(a, n)\) the value of \((a, n)\).

Let us now describe the construction of \(\text{OPT}(p_1, B)\). Let \(p : X_0 \rightarrow X_1 \ldots X_\gamma, \Pi_0 \in \Pi A(X_0)\), and let \(B\) be a block of \(s\)-attributes of \(\Pi_0\), let also \(\Omega(p, \Pi_0) = \langle \Pi_1, \ldots, \Pi_r \rangle\).

Consider the graph \(g = D(p - 0)[\Pi_0, \ldots, \Pi_r]\); by \(\text{SUB}(g, (B, 0))\) we denote the subgraph of \(g\) connected to \((B, 0)\) (cf. Section 1). There are three types of nodes in \(\text{SUB}(g, (B, 0))\):

1. nodes \((A, j)\) where \(j \in [1, \gamma]\) and \(A\) contains \(i\)-attributes, or \(j = 0\) and \(A = B\),
2. nodes \((A, j)\) where \(j \in [1, \gamma]\) and \(A\) contains \(s\)-attributes,
3. nodes \((A, 0)\) where \(A\) contains \(i\)-attributes.

For each node \((A, j)\) of type (1) or (2) we will define a sequence of instructions \(\text{INS}(A, j)\), called an instruction package, as follows:

**Nodes of type (1).** \(\text{INS}(A, j)\) consists of instructions for evaluating the semantic rules of \(p\) that define the attributes \((a, j)\) with \(a \in A\). More precisely, if, for each \(a \in A, (a, j) = f((b_1, i(1)), \ldots, (b_\gamma, i(\gamma)))\) is the corresponding semantic rule of \(p\), then \(\text{INS}(A, j)\) contains the following assignment:

\[\text{value}(a, nj) := f(\text{value}(b_1, ni(1)), \ldots, \text{value}(b_\gamma, ni(\gamma))).\]

The order of these assignments in \(\text{INS}(A, j)\) is irrelevant.
Nodes of type (2). \( \text{INS}(A, j) = \text{eval} \, A(nj) \).

Let \( T(g, (B, 0)) \) be the concatenation of \( \text{INS}(B_1, j_1), \ldots, \text{INS}(B_k, j_k) \), where \( (B_1, j_1), \ldots, (B_k, j_k) \) is a sequence of all the nodes of type (1) and (2) that respects \( \text{SUB}(g, (B, 0)) \). Finally, from \( T(g, (B, 0)) \) one obtains \( \text{OPT}(p, B) \) by the following step.

**Optimization step.** Let \( II_0 = \langle M_0, \rightarrow \rangle \) and let \( B_1, \ldots, B_q \) be the blocks of \( s \)-attributes of \( M_0 \) such that \( B_h \rightarrow^* B \) for each \( h \in [1, q] \); \( \text{OPT}(p, B) \) is obtained by eliminating from \( T(g, (B, 0)) \) every instruction package that appears also in a \( T(g, (B_h, 0)) \).

Observe that this case corresponds to the fact that \( \text{SUB}(g, (B, 0)) \) and \( \text{SUB}(g, (B_h, 0)) \) have a subgraph in common. Intuitively, one can eliminate the above specified instruction packages from \( T(g, (B, 0)) \) because, if \( B_h \rightarrow^* B \), then the ANC-evaluator, before executing a call 'eval \( II_B(n) \)' will always have executed at least one call 'eval \( II_0B_h(n) \)'.

### 3.1.2. The main program

Let \( II_Z = \langle M_Z, \rightarrow \rangle \) and let \( \langle B_1, \ldots, B_n \rangle \) be a sequence of the blocks of \( s \)-attributes of \( M_Z \) that respects \( II_Z \). The main program is as follows:

```plaintext
\begin{verbatim}
begin
  for i := 1 to n do eval II_ZB_i(root of i)
end;
\end{verbatim}
```

This ends Construction 1.

**Example 3.1 (ANC-evaluator).** Let us construct an ANC evaluator for the AG GEX studied in Examples 2.3, 2.6 and 2.8. This evaluator will be based on the good pop-assignment \( II_A \) for GEX given in Example 2.3. The following pop-function \( \Omega \) for \( II_A \) and \( G \) will also be used:

\[
\Omega(1, II_Z) = \langle II_1, II_Z \rangle, \quad \Omega(2, II_Z) = \langle II_Y, II_Z \rangle, \quad \Omega(2, II_Z) = \langle II_Y, II_2 \rangle,
\]

and, clearly,

\[
\Omega(3, II_1) = \Omega(3, II_2) = \Omega(4, II_Y) = \langle \rangle.
\]

The ANC evaluator is described below:

1. From the graph \( g'_Z = D(1-0)[II_Z, II_1, II_Z] \) (\( g_3 = D(p)[II_0, \ldots, II_7] \) is given in Fig. 3), one constructs easily the graph \( \text{SUB}(g'_Z, (\{d, 0\}, 0)) \) and from this graph, it is straightforward to obtain the procedure \( \text{eval} II_Zd. \text{SUB}(g'_Z, (\{d, 0\}) \) and \( \text{eval} II_Zd \) are shown in Figs. 8(a) and (b), respectively.

2. Consider now the graph \( g'_4 = D(2-0)[II_Z, II_Y, II_2] \) (see Fig. 5 for \( g_4 \)). From the two subgraphs \( \text{SUB}(g'_4, (\{s_1, s_1\}, 0)) \) and \( \text{SUB}(g'_4, (\{s_2, 0\}) \) one constructs easily the two procedures shown in Figs. 9(a) and (b). One obtains similar procedures from the graph \( D(2-0)[II_1, II_Y, II_1] \).
procedure eval II,d(n);
begin
INS({i_1}, 1);
eval \Pi_1 s_1 (n1);
INS({i_2}, 2);
eval \Pi_2 s_2 (n2);
INS({i_2}, 1);
eval \Pi_1 s_2 (n1);
INS({i_1, i_2}, 2);
eval \Pi_2 s_1 s_2 (n2);
INS({i_1}, 1);
eval \Pi_1 s_1 (n1);
INS([{d}, 0])
end;

(b)

Fig. 8.

(3) The main program of the ANC evaluator consists simply of the call
'eval \Pi_2 d (root of \textit{t})', where \textit{t} is any complete derivation tree of GEX.

The attentive reader may have noticed that the two procedures of Fig. 9 both do
the evaluation of the attributes of \textit{Y}. Thus, these attributes are evaluated twice! We
will discuss this phenomenon later. Let us first show that Construction 1 is correct.

It is easy to see that the ANC evaluators compute the attributes 'by need', that
is, such an evaluator, for an input tree \textit{t} has as goal to compute the s-attributes of
procedure eval $P_2(s_1, s_3)(n)$;
begin
  case production applied at $n$ of
  2: INS($\{i\}$, 1);
     eval $P_2(s_1, s_3)(n1)$;
     INS($\{i_1, i_3\}$, 2);
     eval $P_2(s_1, s_3)(n2)$;
     INS($\{s_1, s_3\}$, 0);
  3: INS($\{s_1, s_3\}$, 0);
  end of case;
end;

(a)

procedure eval $P_2(s_2)(n)$;
begin
  case production applied at $n$ of
  2: INS($\{i\}$, 1);
     eval $P_2(s_1, s_3)(n1)$;
     INS($\{i_2\}$, n2);
     eval $P_2(s_2)(n2)$;
     INS($\{s_2\}$, 0);
  3: INS($\{s_2\}$, 0);
  end of case;
end;

(b)

Fig. 9

the root of $t$ and it certainly evaluates the attributes of $t$ that are needed to meet this goal, but the other attributes of $t$ (called useless in the sequel) may be left unevaluated. The uncertainty of the ‘may be’ is due to the fact that the ANC evaluator computes blocks of attributes instead of single attributes: in this way useless attributes may be computed simply because they are in a block together with useful ones. Notice that the ANC evaluator of Example 3.1 evaluates all attributes of all complete derivation trees of GEX.

Let us be more formal. For an AG $G$ and a derivation tree $t$ of $G$, an attribute $(a, n)$ of $t$ is useful in $t$ if $(a, n)$ is not an i-attribute of the root of $t$, and if it is in the subgraph of $D(t)$ connected to some s-attribute of the root of $t$. If $(a, n)$ is not useful, then it is useless in $t$.

In what follows two facts will be shown:

(i) the ANC evaluator for an ANC AG $G$, constructed as described in Construction 1, computes for each complete derivation tree its consistent valuation such that (at least) all the useful attributes are defined in it;

(ii) it is easy to modify the ANC evaluators in such a way that the obtained evaluators compute all attributes of their input trees (and not only the useful ones). These new evaluators are called complete ANC evaluators.
For proving (i) some new notions are needed. Consider the computation of any evaluator \( P \) on a tree \( t \) (and, thus, in particular, of an ANC evaluator). We say that \( P \) blocks on \( t \) if it 'tries' to evaluate an attribute \((a, n)\) of \( t \) when the attributes on which \((a, n)\) depends have not yet been computed. If \( P \) never blocks on \( t \), it is successful on \( t \).

For any po-partition \( \Pi = \langle M, \rightarrow \rangle \) of the attributes of a nonterminal \( X \), a correct sequence of \( \Pi \) is a sequence \( S = \langle B_1, \ldots, B_n \rangle \) whose elements are blocks of \( s \)-attributes in \( M \) and which satisfies the following two conditions:

(i) \( S \) respects \( \Pi \),

(ii) for every \( i \in [1, n] \), for every block \( B \) of \( s \)-attributes such that \( B \rightarrow^* B_i \), there must be a \( j < i \) such that \( B_j = B \).

Note that we allow a block to appear more than once in \( S \). Note also that any prefix of \( S \) is also a correct sequence of \( \Pi \).

For a block of \( s \)-attributes \( B \in M \), the input blocks of \( B \) in \( \Pi \) are all the blocks of \( i \)-attributes \( A \in M \) such that there is at least one path in \( \Pi \) from \( A \) to \( B \) and every such path does not traverse any other block of \( s \)-attributes \( B' \) of \( \Pi \) such that \( B' \rightarrow^* B \).

The correctness of Construction 1 is based on the following technical result.

**Lemma 3.2.** For an ANC AG \( G \) and a good pop-assignment \( \Pi A \) for it consider a production \( p : X_0 \rightarrow X_1 \ldots X_r \) of \( G \) and let \( g = D(p - 0)[\Pi_0, \ldots, \Pi_r] \), where \( \Pi_i \in \Pi A(X_i), i \in [0, r] \), such that it satisfies Definition 2.7. Let us fix the following notation:

(i) \( S = \langle B_1^0, \ldots, B_k^0 \rangle \) is a correct sequence of \( \Pi_0 \),

(ii) Seq = INS, \( \ldots, \) INS, is the sequence of instruction packages obtained by concatenating the sections \( OPT(p, B_i^0), \ldots, OPT(p, B_k^0) \) of the procedures \( \text{eval} \Pi_0 B_i^0, \ldots, \text{eval} \Pi_0 B_k^0 \).

The following holds:

1. For any \( i \in [1, k] \), \( \exists r \in [1, q] \), such that \( \text{INS}_r = \text{INS}(B_i^0, 0) \).
2. For any \( j \in [1, q] \) consider in \( \text{Seq} \) the maximal sequence \( \text{INS}_{i(1)}, \ldots, \text{INS}_{i(h)} \), where \( i(1) < i(2) < \cdots < i(h) \), such that for each \( r \in [1, h] \), \( \text{INS}_{i(r)} = \text{INS}(D_i, j) \), where \( D_i \) is a block of \( s \)-attributes of \( \Pi_i \) (hence, \( \text{INS}_{i(r)} = \text{eval } \Pi_i D_i \)). The sequence \( S(j) = \langle D_1, \ldots, D_h \rangle \) is a correct sequence of \( \Pi_j \).
3. For any \( r \in [1, q] \), let \( \text{INS}_r = \text{INS}(A, j) \), if \((A', h)\) is a predecessor of \((A, j)\) in \( g \); then either \( h = 0 \) and \( A' \) is a block of \( i \)-attributes of \( \Pi_0 \), or there is some \( l < r \) such that \( \text{INS}_l = \text{INS}(A', h) \).
4. From (3) it follows that for any \( r \in [1, q] \), such that \( \text{INS}_r = \text{INS}(B, j) \) where \( B \) is a block of \( s \)-attributes of \( \Pi_j \), for each input block \( B' \) of \( B \) in \( \Pi_j \), there exists an \( l \) such that \( \text{INS}_l = \text{INS}(B', j) \).

**Proof.** Immediate from Construction 1. \( \square \)

**Theorem 3.3.** Let \( G \) be an ANC AG and \( P \) an ANC evaluator for \( G \) constructed as described in Construction 1. For any complete derivation tree \( t \) of \( G \), \( P \) computes a consistent valuation of \( t \) such that (at least) all the useful attributes of \( t \) are defined in it.
Proof. Let $\Pi A$ be a good pop-assignment for $G$ and $\Omega$ a pop-function for $G$ and $\Pi A$. Assume that $P$ is constructed using $\Pi A$ and $\Omega$. Since, by Construction 1, $P$ computes the attributes of any input tree $t$ using the appropriate semantic rules, for showing the theorem it suffices to show that $P$ computes all the useful attributes of $t$, and, for showing this, it is sufficient, by the definition of useful attributes, to prove that $P$ is successful on $t$. To this end the following fact is proved.

**Fact 3.4.** For any derivation tree $t$ of $G$ with root labeled by, say $X_0$, and for any pop-partition $\Pi_0 \in \Pi A(X_0)$, let $S = \langle B_1^0, \ldots, B_k^0 \rangle$ be a correct sequence of $\Pi_0$ and let $f$ be an arbitrary function such that for each $a \in IATT(X_0)$, $f(a) \in V(a)$; the following program $P(S)$ is successful on $t$:

$$P(S): \begin{array}{l}
\text{begin} \\
\text{for } i := 1 \text{ to } k \text{ do} \\
\text{begin} \\
\text{for each } (a, n0) \text{ such that } a \text{ belongs to an input block of } B_i^0 \\
\text{in } \Pi_0 \text{ do } \text{value}(a, n0) := f(a); \\
\text{eval } \Pi_0 B_i^0(\text{root of } t) \\
\text{end}; \\
\text{end;}
\end{array}$$

Proof. The proof is by induction on the height of derivation trees of $G$.

**Base.** A tree $t$ consists of one terminal production $p: X_0 \rightarrow w$ only. Since the only useful attributes of such a $t$ are the s-attributes of its root, Fact 3.4 holds by Lemma 3.2(1) from the fact that $S$ is a correct sequence, and that $D(p)[\Pi_0]$ satisfies Definition 2.7.

**Induction step.** Consider a tree $t = n0(t_1, \ldots, t_y)$, where the top-production is $p: X_0 \rightarrow X_1 \ldots X_y$. We assume that Fact 3.4 holds for $t_1, \ldots, t_y$. A second induction is needed. Assume the terminology of Lemma 3.2, and consider the sequence $\text{Seq} = I\text{NS}_1, \ldots, I\text{NS}_q$ of instruction packages defined there. We want to show that for each $j \in [1, q]$, $I\text{NS}_j$ is executed successfully by $P(S)$. Assume that $I\text{NS}_j = I\text{NS}(A, i)$. Two cases are distinguished:

**Case 1.** $(A, i)$ is a node of type (1) of $g$. Consider any direct predecessor $(A', h)$ of $(A, i)$ in $g$. There are two cases:

- (i) $h \neq 0$: by Lemma 3.2(3) and by the second induction hypothesis, $I\text{NS}(A', h)$ has already been executed successfully,
- (ii) $h = 0$: let $I\text{NS}$ be in $\text{OPT}(p, B_i^0)$; since $g$ satisfies Definition 2.7 and since $S$ is a correct sequence of $\Pi_0$, $(A', h)$ is an input block of a block $B_i^0$ of $S$ with $1 \leq r$ and, thus, see $P(S)$, all the attributes in $(A', n0)$ are defined.

This shows that the execution of $I\text{NS}(A, i)$ is successful.

**Case 2.** $(A, i)$ is a node of type (2) of $g$. Consider the prefix $S' = \langle D_1, \ldots, A \rangle$ of the sequence $S(i)$, see Lemma 3.2(2), that corresponds to the instruction packages $I\text{NS}_1, \ldots, I\text{NS}_i$ of $\text{Seq}$. By Lemma 3.2(2), $S'$ is a correct sequence of $\Pi_j$. Lemma
3.2(4) guarantees, by the second induction hypothesis, that one can apply the first induction hypothesis. Thus, \( P(S') \) is successful with input tree \( t_i \) and, hence, the execution of \( INS(A,j) \) is successful too.

This proves Fact 3.4 and, hence, Theorem 3.3: \( P(S) \) coincides with the main program of the ANC evaluator when \( t \) is complete and \( S \) contains all the blocks of \( s \)-attributes of \( \Pi_Z \). □

Let us describe how complete ANC evaluators are built.

3.2. Construction 2: The complete ANC evaluator

Consider an ANC AG \( G \) and a good pop-assignment \( IIA \) for \( G \). A complete ANC evaluator \( P \) for \( G \) contains all the procedures of a normal ANC evaluator for \( G \), plus some new ones. The computation of \( P \) on an input tree \( t \) consists of two phases: in the first one it computes all the attributes that the normal ANC evaluator would compute, and in the second one, in which the new procedures enter in use, it computes the remaining attributes of \( t \).

For each nonterminal \( X_0 \) and po-partition \( \Pi_0 \in IIA(X_0) \), \( P \) has a new procedure ‘eval \( IIA(X_0) \)’ that has the same structure as the procedures described for Construction 1 (cf. Fig. 7), and where each \( OPT(p_i, \emptyset) \) is as follows: let \( p_i : X_0 \rightarrow X_1 \ldots X_r \) and \( g = D(p_i, \emptyset)[\Pi_0, \ldots, \Pi_r] \). Consider all nodes \( (A,j) \) of \( g \) such that there is no path from \( (A,j) \) to a block \( (B,0) \) of \( s \)-attributes of \( X_0 \). There may be two types of such nodes:

1. nodes \( (A,j) \) where \( j \in [1, \gamma] \), and
2. nodes \( (A,0) \) where \( A \) is a block of \( i \)-attributes.

Let \( S = ((A_1,j_1), \ldots, (A_k,j_k)) \) be any sequence of the nodes of type (1) respecting \( g \), then CODE1 is the concatenation of \( INS(A_1,j_1), \ldots, INS(A_k,j_k) \) as defined in Construction 1. Let CODE2 be the following sequence of instructions: eval \( IIA(A_1,j_1)(\emptyset(n_1)); \ldots; \) eval \( IIA(A_k,j_k)(\emptyset(n_\gamma)); OPT(p_i, \emptyset) \) is equal to CODE1;CODE2.

The main program of the complete ANC evaluator is obtained from that of the normal ANC evaluator by adding to it a recursive call to a new procedure that starts the second phase of the computation:

\[
\text{begin}
\text{for } i := 1 \text{ to } k \text{ do eval } II_B_i(\text{root of } t); \text{ eval } II_A(\text{root of } t)
\text{end;}
\]

This ends Construction 2.

Lemma 3.5. Let \( G \) be an ANC AG and \( P \) a complete ANC evaluator for \( G \) built as described in Construction 2. For any complete derivation tree \( t \) of \( G \), \( P \) computes a consistent full valuation of \( t \).

Proof. In a way similar to the proof of Fact 3.4 one can prove the following fact and hence the lemma. □
Fact 3.6. Consider the derivation tree $t$ and the program $P(S)$ of Fact 3.4. Assume that $S = (B'_1, \ldots, B'_k, 0)$, where $B'_1, \ldots, B'_k$ are all the blocks of $s$-attributes of $\Pi_o$ and that the input blocks of $\emptyset$ are those blocks of $i$-attributes of $\Pi_o$ that are input blocks of no $B'_i, i \in [1, k]$. $P(S)$ evaluates all attributes of $t$.

Let us now turn our attention to the fact that ANC evaluators may compute some attributes several times (clearly, assigning to them the same values each time). This phenomenon is studied in the following example.

Example 3.7 (the recomputation problem). As already remarked the ANC evaluator of Example 3.1 does recompute some attributes: both eval $I_2 s_1 s_3$ and eval $I_2 s_2$ contain a call to eval $I_1 s$ (cf. Fig. 9). This is also true for the three procedures corresponding to the po-partition $\Pi_1$. The reason for this is that $\Pi_1$ and $\Pi_2$ do not give any information about the order in which the procedures eval corresponding to their blocks of $s$-attributes are executed. Thus, for instance, considering $\Pi_2$ one does not know whether, for any node $n$ of an input tree such that $\Pi_2$ is associated to $n$, eval $I_2 s_1 s_3(n)$ is executed before or after eval $I_2 s_2(n)$. For this reason both procedures must contain the call to eval $I_1 s(n)$.

The notion of po-partition associated to a node $n$ can be made precise as follows: consider an ANC AG $G$, a good pop-assignment $\Pi A$ for it and a corresponding pop-function $\Omega$. For any complete derivation tree $t$ of $G$, the po-partition $\Pi$ associated to a node $n$ of $t$ is defined recursively as follows:

(i) if $n$ is the root of $t$, then $\Pi = \Pi_2$.
(ii) otherwise, let $m$ be the father of $n$ and let $n = mj, j \geq 1$, if $\Pi_0$ is the po-partition associated to $m$ and $p$ is the production applied at $m$, then $\Pi = \Pi_j$, where $\Omega(p, \Pi_0) = \langle \Pi_1, \ldots, \Pi_r \rangle$.

The first idea for solving the recomputation problem is that of fixing an order among the blocks of $s$-attributes of the po-partitions of $\Pi A$, obtaining a new pop-assignment $\Pi A'$ that contains the same number of po-partitions as $\Pi A$. In the example under consideration this can be done, but, in general, the pop-assignment $\Pi A'$, necessary for eliminating the recomputation, may contain many (exponentially) more po-partitions than the original one.

Let us first consider our example. Let $\Pi A'$ be the pop-assignment obtained from $\Pi A$ by substituting $\Pi A'(X) = \{\Pi'_1, \Pi'_2\}$ for $\Pi A(X)$, where

$$\Pi'_1 = \{i_1\} \rightarrow \{s_1\} \rightarrow \{i_2\} \rightarrow \{s_2\} \rightarrow \{i_3\} \rightarrow \{s_3\},$$

$$\Pi'_2 = \{i_2\} \rightarrow \{s_2\} \rightarrow \{i_1, i_3\} \rightarrow \{s_1, s_3\}.$$ 

It is easy to see that $\Pi A'$ is good for GEX and that the ANC evaluator based on it does no longer recompute any attribute.

Eliminating the recomputation is not always as simple as in the previous example. Consider, for instance, the AG $G'$ obtained from GEX by adding to it the production shown in Fig. 10. $\Pi A$ is good for $G'$: $D(5)[\Pi_1, \Pi_1] \text{ and } D(5)[\Pi_2, \Pi_2]$ both satis
Definition 2.7. The ANC evaluator for $G'$ based on $IIA$ recomputes the attributes of $Y$ exactly as that of Example 3.1 for $GEX$, but this problem can no longer be solved transforming $IIA$ into $IIA'$: $IIA'$ is no longer good for $G'$! It is easy to see, in fact, that both $D(5)[II_1', II_2']$ and $D(5)[II_3', II_5']$ are circular. For solving the problem one needs to add to $IIA'(X)$ a new po-partition $II_3'$ as follows:

$$II_3' = \{i_3\} \rightarrow \{s_3\} \rightarrow \{i_2\} \rightarrow \{s_2\} \rightarrow \{i_1\} \rightarrow \{s_1\},$$

that is used in the following way:

$$\Omega(5, II_1') = \langle II_1' \rangle, \quad \Omega(5, II_3') = \langle II_3' \rangle,$$
$$\Omega(5, II_2') = \langle II_2' \rangle, \quad \Omega(2, II_3') = \langle II_1', II_3' \rangle.$$

Theorem 2.12 shows that for any ANC AG $G$ one can find a top-assignment that is good for $G$ and, hence, an ANC evaluator without recomputation. It should be understood, however, that, for eliminating the recomputation, it is not always necessary to find a top-assignment: it may very well be that a pop-assignment suffices. In general, for a given ANC AG $G$ there is a whole range of good pop-assignments. At one extreme of this range are the trivial IS pop-assignments and at the other extreme the top-assignments. The former contain the minimum amount of information about the attribute dependencies that is necessary for constructing ANC evaluators, whereas the latter contain more of such information and, therefore, the corresponding evaluators do not compute. Unfortunately, there are ANC AGs such that all good top-assignments for them have exponential size w.r.t. the AG and w.r.t. some good pop-assignment for them. Because of this, it is interesting to be able to find, for any given ANC AG, an 'intermediate' good pop-assignment (w.r.t. the two extremes defined above) which realizes a 'good' compromise between the size and the recomputation of the corresponding ANC evaluator.

This problem has been studied in [12] where an algorithm is given for deciding whether a given ANC evaluator (noncomplete) recomputes some attribute. This algorithm (that is doubly exponential) gives, in case that recomputation is found, the po-partitions responsible for the phenomenon.

4. The incremental evaluators

In this section it will be shown that one can construct incremental evaluators for DNC AGs in a way essentially similar to that in which ANC evaluators are built.
for ANC AGs. These incremental evaluators are called DNC evaluators and they will be described in Construction 3 below, but let us first explain what one means with incremental attribute evaluation (for more details see [14]).

In syntax-directed editors attributes may be used for checking the static semantics of the developed program. At each moment of the construction of a program, the editor keeps the parse tree \( t \) of the (partial) program \( P \) developed so far, together with a full and consistent valuation of \( t \) (the attributes that cannot be 'really' evaluated due to the incompleteness of \( P \) are given some special value). Any transformation of \( P \), either the addition or the modification of a statement, is viewed by the editor as the replacement of a subtree of \( t \) with a new fully and consistently attributed subtree \( s \). The node \( u \) of \( t \) at which the replacement takes place is called the rep-node. Let \( t' \) be the tree obtained in this way, the following valuation \( \text{VAL}_{t'} \) is associated to \( t' \): let \( \text{VAL}_t \) and \( \text{VAL}_s \) be the valuations of \( t \) and \( s \), respectively,

(i) for every node \( m \) of \( t' \) such that \( m \neq u \) and \( m \) is not a descendant of \( u \), \( \text{VAL}_{t'}(a, m) = \text{VAL}_t(a, m) \) for every attribute of \( m \);
(ii) for each node \( uh \) of \( t' \) with \( h \neq e \), \( \text{VAL}_{t'}(a, uh) = \text{VAL}_t(a, h) \);
(iii) for the rep-node \( u \), for each \( i \)-attribute \( (a, u) \), \( \text{VAL}_{t'}(a, u) = \text{VAL}_t(a, 0) \), and for each \( s \)-attribute \( (b, u) \), \( \text{VAL}_{t'}(b, u) = \text{VAL}_s(b, u) \).

In general, some of the attributes of \( u \) have inconsistent values in \( \text{VAL}_{t'} \) (whereas all other attributes are consistent: this is the reason for associating such a \( \text{VAL}_{t'} \) to \( t' \)). The task of the incremental evaluator is that of transforming \( \text{VAL}_{t'} \) into a consistent valuation by reevaluating only some of the attributes of \( t' \). Clearly, incremental evaluators may be more or less efficient, depending on how many attributes of \( t' \) they reevaluate w.r.t. the minimum number of reevaluations necessary. An incremental evaluator is said to be optimal if, for every input tree \( t' \), it performs a number of operations linear in the number of attributes of \( t' \) whose values in the consistent valuation of \( t' \) is different from what they have in \( \text{VAL}_{t'} \).

4.1. Construction 3: The DNC evaluator

We distinguish two parts of this construction. In Section 4.1.1 we fix the notation and the concepts to be used in Section 4.1.2, where the program describing the evaluator is given.

4.1.1. Preliminaries

Let \( G \) be a DNC AG and \( \Pi A \) a fine pop-assignment for \( G \). Let also \( \Psi \) be a function as follows: for any production \( p : X_0 \rightarrow X_1 \ldots X_y \), po-partition \( \Pi \in \Pi A(X_i) \), \( i \in \{0, \gamma\} \), \( \Psi(p, \Pi, j) = (\Pi_0, \ldots, \Pi_y) \), such that, for every \( i \in \{0, \gamma\} \), \( \Pi_i \in \Pi A(X_i) \), \( \Pi_j = \Pi \), and \( D(p)[\Pi_0, \ldots, \Pi_y] \) satisfies Definition 2.9. Such a function \( \Psi \) is called a fine pop-function for \( G \) and \( \Pi A \).

The DNC evaluators have procedures traversing the input tree in a top-down fashion (essentially similar to the procedures of the complete ANC evaluators of Construction 2) and procedures that traverse the tree in a bottom-up fashion. These last procedures have the task of computing blocks of \( i \)-attributes. In what follows
we will distinguish these two types of procedures by calling them *top-down* and *bottom-up* procedures, respectively.

Using these two types of procedures, the DNC evaluator can traverse several times and in both directions an occurrence $\langle n_0, \ldots, n_\gamma \rangle$ of a production $p: X_0 \rightarrow X_1 \ldots X_\gamma$ in the input tree, i.e., from $n_0$ to an $n_j$ and from an $n_j$ to $n_0$. Let $\Pi$ be the po-partition associated to $n_0$ in the first case and $\Pi'$ that associated to $n_j$ in the second case. If in both cases the function $\Psi$ is used for deciding what sequence of po-partitions must be associated to the occurrence of $p$, an error may occur: it may, in fact, be that $\Psi(p, \Pi, 0) \neq \Gamma(p, \Pi', j)$. If this would be the case, at different moments of the computation, different po-partitions sequences would be associated to the same occurrence of $p$. Clearly, we must avoid this possibility: every occurrence of a production must be associated to only one sequence of po-partitions throughout the evaluation process.

To this end we choose to do the following: whenever the occurrence of a production is encountered for the first time by the evaluator, the sequence of po-partitions associated to it is fixed and this sequence will be used at all other visits to that occurrence. More precisely, every internal node $n$ of the input tree is given a Boolean variable $\text{vis}(n)$ that is true iff $n$ has been already visited by the evaluator; hence, initially, all these variables are set to false. Each internal node $n$ will have also two variables: $\text{above}(n)$ and $\text{below}(n)$; both will contain values of the form $[p, \langle \Pi_0, \ldots, \Pi_\gamma \rangle]$. In the case of $\text{above}(n)$ $p$ is the production applied to the father of $n$ and $\langle \Pi_0, \ldots, \Pi_\gamma \rangle$ the sequence of po-partitions that has been fixed for this occurrence of $p$ (according to $\Psi$); for $\text{below}(n)$, clearly, $p$ is the production applied at $n$ and $\langle \Pi_0, \ldots, \Pi_\gamma \rangle$ the sequence fixed for it.

A node $n$ can be entered by the evaluator from above (top-down procedures) or from below (bottom-up procedures). In both cases the evaluator carries the po-partition $\Pi$ associated to $n$ ($\Pi \in \Pi A(\text{lab}(n))$). Let us describe what must be done in the two cases if $\text{vis}(n)$ is false:

*Case 1:* from above.

```plaintext
procedure topbo($\Pi$, $n$); begin
    case production applied at $n$ of $p_1$; $\ldots$ $p_\iota$: $\ldots$
        $p_i$: $\text{below}(n) := [p_i, \Psi(p_i, \Pi, 0)];$
        for $j := 1$ to $\gamma$ do
            $\text{above}(nj) := \text{below}(n);
            allocate space for and initialize the variables associated to the nodes $n_1, \ldots, n_\gamma$;
        end;
    end of case;
end;
```
Case 2: from below.

procedure botop(II, nj);
begin
  case production applied at n of
    p₁; ...;
    ...;
    pᵢ: below(n) := [pᵢ, Ψ(pᵢ, II, j)];
    for k := 1 to γ do
      above(nᵏ) := below(n);
      allocate space for and initialize the variables associated to the nodes
      n₀, ..., n₋₁, n₊₁, ..., nγ;
    ...;
  end of case;
end.

Consider a node nj such that above(nj) and below(nj) are both defined. The DNC evaluator is such that, if above(nj) = (II₀, ..., IIᵢ) and below(nj) = (II₀', ..., IIᵢ'), then IIᵢ = II₀ and IIᵢ₊₁ ∈ II(A(\text{lab}(nj))). This po-partition is said to be associated to nj by the evaluator. Clearly, the po-partition associated to nj is defined whenever either above(nj) or below(nj) is defined.

The DNC evaluator needs to have some Boolean variables associated to every node that it visits. For any node n, if II = (M, →) is the po-partition associated to it, then n must have two Boolean variables tr(B, n) and mod(B, n) for each block B ∈ M: tr(B, n) is true iff the block (B, n) of attributes of n has been already recomputed by the evaluator, and mod(B, n) is true iff the new value of at least one attribute of (B, n) is different from its old value. The procedures topbo and botop, given above, take care of allocating the space for these variables when they are needed and of initializing them to false.

For performing the recomputation of potentially inconsistent attributes, the DNC evaluator uses the Boolean function update given in Fig. 11. Clearly, the update function returns true iff some attribute of (B, nj) has changed value.

4.1.2. The evaluator

As already mentioned, the DNC evaluator has two types of procedures: top-down and bottom-up. Since the top-down ones are essentially similar to the procedures of the complete ANC evaluator, we will consider first the bottom up ones.

The general form of a bottom-up procedure is given in Fig. 12. It is important to notice the similarity between these procedures and those of the ANC evaluator (compare Fig. 12 with Fig. 7). This is due to the fact that the same ideas are at the base of all Constructions 1-3. The outer case statement of the procedure of Fig. 12 takes the place of that for individuating the production applied at n of the procedure of Fig. 7; the inner case statement individuates the position of nk in the right-hand
function update(\(B\), \(n_j\), SEM): boolean;
\(B\) is a block of attributes, \(n_j\) the node interested by the recomputation, and SEM a set of semantic rules

begin
update := false;
for every \(a \in B\) do
begin
let \((q_j) = f((a_1, i(1)), \ldots, (a_q, i(q)))\) be the semantic rule of SEM that defines \((a, j)\);
newvalue := \(f(\text{value}(a_1, ni(1)), \ldots, \text{value}(a_q, ni(q)))\);
if value\((a, n_j) \neq \text{newvalue}\) then
begin
value\((a, n_j) := \text{newvalue};\)
update := true
end;
end;
end;

Fig. 11.

side of this production. Clearly, in Fig. 12, in the name of the procedure, \(\Pi \subset \Pi A(\text{lab}(n_j))\) and \(B\) is a block of \(i\)-attributes of \(\Pi\). \(\text{OPT}([p_i, h_i], (B, j))\) is constructed as follows: let \(p = p = X_0 \rightarrow X_1 \ldots X_\gamma\) and \(h_i = \langle II_0, \ldots, II_\gamma\rangle\), let also \(g = D(p-j)[II_0, \ldots, II_\gamma];\) with \(\text{SUB}(g, (B, j))\) we denote the subgraph of \(g\) connected to \((B, j)\). There are four types of nodes in \(\text{SUB}(g, (B, j))\):

1. nodes \((A, i)\), where either \(i \in [1, \gamma], i \neq j\) and \(A\) is a block of \(i\)-attributes, or \(i = 0\) and \(A\) is a block of \(s\)-attributes;

procedure eval \(II B(nk)\);
begin
if not(vis\((nk)\)) then
begin
botop(\(II, nk)\);
vis\((nk) := \text{true}
end;
end;
case above\((nk)\) of
\([p_1, h_1]\): 
\[p_i, h_j]:\ case k of
\(1: \ldots\)
\(j: \text{OPT}([p_i, h_j], (B, j));\)
\(\ldots\)
end of case;
end of case;
end;

Fig. 12.
(2) nodes \((A, i)\), where either \(i \in [1, \gamma], i \neq j\) and \(A\) is a block of \(s\)-attributes, or \(i = 0\) and \(A\) is a block of \(i\)-attributes;

(3) the node \((B, j)\);

(4) nodes \((A, j)\), where \(A\) is a block of \(s\)-attributes.

For each node \((A, i)\) of types (1)–(3) we define an instruction package \(\text{INS}(A, i)\), as follows:

*For a node \((A, i)\) of type (1).*

\[
\text{INS}(A, i): \quad \text{if not}(\text{tr}(A, ni)) \text{ then}
\]

\[
\begin{align*}
&\text{begin} \\
&\quad \text{if } \exists \text{ an immediate predecessor } (A', k) \text{ of } (A, i) \text{ in } \text{SUB}(g, (B, j)) \text{ such that } \text{mod}(A', k) = \text{true} \\
&\quad \text{then } \text{mod}(A, ni) := \text{update}(A, ni, \text{SEM}(p)); \\
&\quad \text{tr}(A, ni) := \text{true} \\
&\text{end};
\end{align*}
\]

*For a node \((A, i)\) of type (2).*

\[
\text{INS}(A, i): \quad \text{if not}(\text{tr}(A, ni)) \text{ then}
\]

\[
\begin{align*}
&\quad \text{if } \exists \text{ an immediate predecessor } (A', k) \text{ of } (A, i) \text{ in } \text{SUB}(g, (B, j)) \text{ such that } \text{mod}(A', nk) = \text{true} \\
&\quad \text{then } \text{eval } II_A(ni); \quad \{\text{the procedure takes care of } \text{tr}(A, ni) \text{ and } \text{mod}(A, ni)\} \\
&\quad \text{else } \text{tr}(A, ni) := \text{true};
\end{align*}
\]

*For the node \((B, j)\) of type (3).*

\[
\text{INS}(B, j): \quad \text{if } nj \text{ is the rep node}
\]

\[
\begin{align*}
&\text{then} \\
&\quad \text{begin} \\
&\quad \quad \text{mod}(B, nj) := \text{update}(B, nj, \text{SEM}(p)); \\
&\quad \quad \text{tr}(B, nj) := \text{true} \\
&\quad \text{end}; \\
&\text{else} \\
&\quad \text{begin} \\
&\quad \quad \text{if } \exists \text{ an immediate predecessor } (A', k) \text{ of } (B, j) \text{ in } \text{SUB}(g, (B, j)) \text{ such that } \text{mod}(A', nk) = \text{true} \\
&\quad \quad \text{then } \text{mod}(B, nj) := \text{update}(B, nj, \text{SEM}(p)); \\
&\quad \quad \text{tr}(B, j) := \text{true} \\
&\quad \text{end}.
\end{align*}
\]

\(\text{OPT}([p, h,], (B, j))\) is obtained in two steps (as in Construction 1) as follows:

*Step 1.* One constructs a sequence \(T([p, h,], (B, j))\) of instructions by concatenating the instruction packages \(\text{INS}(A, i)\) of every node of \(\text{SUB}(g, (B, j))\) of type (1), (2) or (3) in any order that respects \(\text{SUB}(g, (B, j))\).
Step 2. An optimization step is performed: let $II = (M, \rightarrow)$ and let $B_1, \ldots, B_k$, $k \geq 0$, be all blocks of $i$-attributes of $M$ such that, for each $i \in \{1, k\}$, $B_i \neq B$ and $B_i \rightarrow^* B$, $OPT([p, h_1], (B, j))$ is obtained from $T([p, h_1], (B, j))$ by eliminating from it any instruction package $INS(A, i)$ that is already contained in $T([p, h_1], (B, j))$, $f \in [1, k]$.

One should notice the fact that for a node of type (3) in $INS(B, j)$ one considers separately the case that $nj$ is the rep-node and, if this is so, $(B, nj)$ is recomputed directly, i.e., without testing the mod-variables of its predecessors. This behaviour is justified by the way the valuation $VAL_{i'}$, associated to $i'$, is defined for $u$, see the beginning of the section. For any attribute $(a, u)$, $VAL_{i'}(a, u)$ has not been computed (in general) with the semantic rule defining $(a, u)$ in $t'$ and, thus, it must be recomputed even though all attributes on which $(a, u)$ depends have already been treated and their values are unchanged. Notice that it is precisely with the reevaluation of some attributes of $u$ that the whole evaluation process starts.

Let us now turn to the top-down procedures. As for the complete ANC evaluator, the DNC evaluator contains two types of top-down procedures: those that have the task of computing a block of $s$-attributes of the node that is passed as parameter (see Construction 1), and those that are used for completing the evaluation (see Construction 2). In what follows these two types of procedures are called top-down1 and top-down2, respectively.

The construction of the top-down1 procedures should present no real difficulty. Roughly, they are obtained from the procedures introduced in Construction 1, see Fig. 7, by performing the following modifications:

1. An initialization part, similar to that of the bottom-up procedures (see Fig. 11), must be added, but the procedure topto must be called in place of botop.

2. The case statement that forms the main part of the body of the procedure must now depend on the value of below(n), instead of on the production applied at n as in Fig. 7, cf. the symmetry with Fig. 11, where above(nj) is used.

3. Instead of the code OPT([p, B]) of Construction 1, OPT([p, h_1], B) must be a sequence of instruction packages constructed as for the bottom-up procedures presented above. Essentially, one needs to add appropriate tests to OPT([p, B]); the graph that must be examined for building OPT([p, h_1], B) is the same as that used for building OPT([p, B]), i.e., SUB(g, (B, 0)), see Construction 1.

The top-down2 procedures are obtained from the procedures introduced in Construction 2 (let us call them $\phi$-procedures) by the following modifications:

(a) The modifications (1)-(3) above must be applied also in this case.

(b) In addition to the parameter $n$ of type node of the input tree, the top-down2 procedures have a second parameter $j \geq 0$. Recall from Construction 2 that the body of a $\phi$-procedure consists of two parts: CODE1 and CODE2. In the first one attributes are evaluated and, in the second one, recursive calls to $\phi$-procedures are executed. The effect of the new parameter $j$ is as follows. If $j = 0$, it has no effect; if $j > 0$, then it has an effect on both CODE1 and CODE2: neither blocks of attributes of $nj$ are reevaluated in CODE1, nor a recursive call to $nj$ is executed in CODE2.
(c) In CODE2, before making a recursive call to a node $ni$, it is convenient to test whether $\text{mod}(B, i)$ is true for at least one of its blocks of $i$-attributes; if this is not the case then the call can be avoided.

In Fig. 13 the main program of the DNC evaluator is given. In this program the following is assumed:
(a) $u$ is the rep-node,
(b) for each nonterminal $X$, $\text{Start}(X)$ is a po-partition that must be used for beginning the evaluation in case $\text{lab}(u) = X$,
(c) $\text{Start}([\text{lab}(u)]) = \Pi = (M, \rightarrow)$ and $(b_1, \ldots, b_k)$ is a sequence of all the elements of $M$ that respects $\Pi$.

Main program

```
begin
  botop(Start(lab(u)), u);
  topbo(Start(lab(u)), u);
  vis(u) := true;
  case Start(u) of
    ...
  end of case;
  m := u;
  repeat
    let $m = nj$;
    $m := n$;
    if not(vis(m)) then
      begin
        (a) let below(m) = $[p, h \rightarrow \Pi_0, \ldots, \Pi_\alpha]$;
           botop($\Pi_0, m$);
           vis(m) := true
      end;
    end of case;
    case below(m) of
      ...
    end;
  until (m is the root of i) or (\forall i \in [1, \alpha] not(mod(B_i, m)));
end;
```

Fig. 13.

Let us explain briefly how the main program works. For this purpose, in Fig. 13 the important sections of the program are marked by distinct numbers. Let us examine them in order:
(1) The nodes of the production occurrences above and below the rep-node $u$ are initialized: above and/or below variables are defined and mod and tr flags are initialized.

(2) According to the po-partition chosen for $u$ one has to:
   (a) evaluate all attributes of $u$,
   (b) complete the incremental attribute evaluation in the subtree rooted in $u$, i.e., the changes of the values of the i-attributes of $u$ are propagated below $u$.

(3) The program moves from $u$ towards the root of $t$ a node at a time, until the root is reached or a node is found in which all attributes are unmodified. For each node $m$ traversed one does:
   (a) initialize the production above $m$ if vis($m$) is false,
   (b) evaluate the attributes of $m$ that have not yet been treated,
   (c) propagate the modifications below $m$ with the exception of the subtree rooted in $m_j$: $m_j$ is on the path from $m$ to $u$ and thus, the subtree rooted in $m_j$ is already fully evaluated.

This ends Construction 3.

The following example illustrates Construction 3.

Example 4.1 (DNC evaluator). Consider the DNC AG $GEX'$ and the fine pop-assignment $\overline{ITA}$, given in Example 2.10. We want to apply Construction 3 in order to obtain a DNC evaluator $E$ for $GEX'$ based on $\overline{ITA}$.

The fine pop-function $\Psi$ for $\overline{ITA}$ that will be used is as follows:

$$\Psi(1, \Pi_Z, 0) = \Psi(1, \Pi_X, 1) = \Psi(1, \Pi_X, 2) = \langle \Pi_Z, \Pi_X, \Pi_X \rangle;$$

$$\Psi(2, \Pi_X, 0) = \Psi(2, \Pi_Y, 1) = \Psi(2, \Pi_X, 2) = \langle \Pi_X, \Pi_Y, \Pi_X \rangle;$$

$$\Psi(3, \Pi_X, 0) = \langle \Pi_X \rangle; \, \Psi(4, \Pi_Y, 0) = \langle \Pi_Y \rangle.$$  

This example illustrates the main points of the DNC evaluator still remaining very simple: each nonterminal node of $GEX'$ has only one possible po-partition and, moreover, $GEX'$ is such that in all its derivation trees all attributes are useful and, therefore, no top-down2 procedures are needed.

We will first give the bottom-up procedures of $E$, then a top-down1 procedure, and, finally, the main program.

The bottom-up procedures of $E$

These procedures are contained in Figs. 14 and 15. The section $\text{OPT}([1, \langle \Pi_Z, \Pi_X, \Pi_X \rangle], \{(i_1, i_3), 1\})$, marked by (*) in Fig. 14, is constructed considering the graph $\text{SUB}(g, \{(i_1, i_3), 1\})$, where $g = D(1-1)[\Pi_Z, \Pi_X, \Pi_X]$. This graph is as follows:

$$\{(i_1, i_3), 1\} \leftarrow \{(s_1, s_3), 2\} \leftarrow \{(i_1, i_3), 2\}.$$  

For constructing the section $S = \text{OPT}([1, \langle \Pi_Z, \Pi_X, \Pi_X \rangle], \{i_2\}, 1)$, marked by (*) in
Fig. 15, one must use the graph SUB$(g, \{i_2\}, 1))$, where $g = D(1-1)[\Pi_Z, \Pi_X, \Pi_X]$. This graph is shown in Fig. 16, together with $\text{SUB}(g, \{i_1, i_3\}, 1))$. Since in $\Pi_X, \{i_1, i_3\} \rightarrow \{i_2\}$, one has applied the optimization step (cf. part 4.1.2 of Construction 3) and the section $S$ is constructed from the reduced graph, $\text{SUB}(g, \{i_2\}, 1)) - \text{SUB}(g, \{i_1, i_3\}, 1)).$

```
procedure eval $\Pi_X i_1 i_3(nk)$;
begin
  if not(vis(nk)) then
    begin
      botop($\Pi_X, nk$);
      vis(nk) := true
    end;
  case above(nk) of
    [1, $\{\Pi_Z, \Pi_X, \Pi_X\}$]:
      case $k$ of
        1: INS($\{i_1, i_3\}$, 2);
        (*) INS($\{s_1, s_2\}$, 2);
        INS($\{i_1, i_3\}$, 1);
        2: INS($\{i_1, i_3\}$, 2);
      end of case;
    [2, $\{\Pi_X, \Pi_Y, \Pi_X\}$]:
      {k must be 2}
      INS($\{i_1, i_3\}$, 0);
      INS($\{i_1, i_3\}$, 2);
  end of case;
end;

Fig. 14.
```

```
procedure eval $\Pi_X i_2(nk)$;
begin
  if not(vis(nk)) then
    begin
      botop($\Pi_X, nk$);
      vis(nk) := true
    end;
  case above(nk) of
    [1, $\{\Pi_Z, \Pi_X, \Pi_X\}$]:
      case $k$ of
        1: INS($\{i_2\}$, 2);
        (*) INS($\{s_2\}$, 2);
        INS($\{i_2\}$, 1);
        2: INS($\{i_2\}$, 1);
        INS($\{s_1, s_2\}$, 1);
        INS($\{i_2\}$, 2);
      end of case;
    [2, $\{\Pi_X, \Pi_Y, \Pi_X\}$]:
      {k must be 2}
      INS($\{s_1, s_2\}$, 0);
      INS($\{i_2\}$, 0);
      INS($\{i_2\}$, 2);
  end of case;
end;

Fig. 15.
```
Fig. 16.

The top-down1 procedure of E

In Fig. 17 the procedure eval $II_X s_1 s_3$ is shown. It should be easy by now to understand how the two OPT-sections of this procedure are built. The remaining top-down1 procedures of E are similar to this one.

The main program of E

This program is shown in Fig. 18. The following notation is assumed: \( \text{Start}(X) = \Pi_X \) and \( \text{Start}(Y) = \Pi_Y \), \( A_1 = \{i_1, i_3\} \), \( A_2 = \{s_1, s_3\} \), \( A_3 = \{i_2\} \), and \( A_4 = \{s_2\} \). The program is simpler than that shown in Fig. 13 because E does not need top-down2 procedures.

```plaintext
procedure eval $II_X s_1 s_3 (n)$;
begin
  if not(vis(n)) then
    begin
      topbo($II_X$, n);
      vis(n) := true
    end;
  case below(n) of
  [2, ($II_X, II_Y, II_X$)]:
    INS($\{i_1, i_3\}$, 2);
    INS($\{s_1, s_3\}$, 2);
    INS($\{i_2\}$, 1);
    INS($\{s_1\}$, 1);
    INS($\{s_2\}$, 0);
  [3, ($II_X$)]:
    INS($\{s_1, s_2\}$, 0);
end of case;
end;
```

Fig. 17.
In what follows we will show that the DNC evaluators are correct and optimal. Throughout the rest of this section the following notation is used.

**Notation 4.2.** $G$ is a DNC AG, $IIA$ a fine pop-assignment for $G$, $\Psi$ a fine pop-function for them, Start a function as explained in the description of the main program in Construction 3 and $P$ a DNC evaluator for $G$; $t'$ is a complete derivation tree of $G$ and $\text{VAL}_r$ an inconsistent full valuation for $t'$ such that all inconsistent attributes in $\text{VAL}_r$ are attributes of the same node $u$ ($u$ is different from the root of $t'$). $\text{NEWVAL}_r$ is the consistent full valuation of $t'$. Intuitively, $t'$ and $\text{VAL}_r$ are obtained from a fully and consistently attributed tree $t$ by a tree-replacement at $u$ (see the beginning of this section).

**The main program of $E$**

```
begin
  botop(Start(lab(u)), u);
  topbo(Start(lab(u)), u);
  vis(u) := true;
  case Start(u) of
    $H_X$: for $i := 1$ to $4$ do eval $H_XA_i(u)$;
    $H_Y$: eval $H_YA_i(u)$;
      eval $H_Ys(u)$;
  end of case;
  $m := u$;
  repeat
    continue := true;
    $m := \text{father}(m)$;
    if $\text{not}(\text{vis}(m))$ then
      begin
        let below(m) = [$p, (II, II, Z)$$]$;
        botop($H_0, m$);
        vis(m) := true
      end;
    case below(m) of
      [$1, (II, II, II, II)$$]$:
        eval $H_d(m)$;
      [$2, (II, II, II, II)$$]$:
        for $i := 1$ to $4$ do
          if $\text{not}(\text{tr}(A_i, m))$ then
            begin
              eval $H_XA_i(m)$;
            end;
          if $\text{mod}(A_i, m)$ then continue := false;
        end;
    end of case;
    until lab(m) = $Z$ or continue
  end;
```

Fig. 18.

Using Notation 4.2, the following notions are introduced. An attribute $(a, n)$ of $t'$ is **to-change** if

$$\text{VAL}_r(a, n) \neq \text{NEWVAL}_r(a, n).$$
For any internal node \( n \) of \( t' \), \( \text{par}(n) \) is the po-partition that \( P \) would associate to \( n \) if it would visit it. Clearly, \( \text{par}(n) \) is uniquely determined by \( \text{Start} \) and \( \Psi \) (see part 4.1.1 of Construction 3).

Consider an occurrence \( \langle n_0, \ldots, n_\gamma \rangle \) of production \( p : X_0 \rightarrow X_1 \ldots X_\gamma \) in \( t' \) and assume that \( \text{par}(n_i) = \Pi_i = (M_i, \rightarrow_i), i \in [0, \gamma] \). Let \( B \) be a block of \( s \)-attributes in \( M_0 \). For \( j \in [0, \gamma] \), a block \( (D, nj) \), where \( D \in M_j \), is owned by \( (B, n_0) \) if there is a path in \( g = D(p - 0)[\Pi_0, \ldots, \Pi_j] \) from \( (D, j) \) to \( (B, 0) \) and no such path exists from \( (D, j) \) to \( (B', 0) \), where \( B' \) is a block of \( s \)-attributes of \( M_0 \) such that \( B' \rightarrow_+ B \).

A block \( (D, nj) \) is at the frontier of \( (B, n_0) \) if it is not owned by \( (B, n_0) \) and if there is an arc in \( g \) from \( (D, j) \) to a block \( (D', i) \) such that \( (D', ni) \) is owned by \( (B, n_0) \). A block is directly linked to \( (B, n_0) \) if it is either owned by \( (B, n_0) \) or it is at the frontier of \( (B, n_0) \).

It is useful to make some remarks about these new notions. Consider the section \( \text{OPT}([p, (\Pi_0, \ldots, \Pi_q)], (B, 0)) \) of the procedure \( \text{eval} \) \( II_0B \) (see Construction 3). Let it be \( \text{Seq} = \text{INS}_1, \ldots, \text{INS}_q \), where each \( \text{INS}_i \) is an instruction package corresponding to a node \( (A, j) \) of \( g \). By considering the optimization step of Construction 3 (see also that of Construction 1) it is not difficult to see that for each \( i \in [1, q] \), \( \text{INS}_i = \text{INS}(A, j) \) iff \( (A, nj) \) is owned by \( (B, n_0) \). One can also see that, since in each \( \text{INS}(A, j) \) one tests the variables mod of the immediate predecessors of \( (A, j) \) in \( g \), such tests in \( \text{Seq} \) concern the variables mod of blocks either owned by \( (B, n_0) \) or at the frontier of \( (B, n_0) \).

A block \( (D, m) \) is linked to \( (B, n_0) \) in \( t' \) if it is in the minimal set \( K \) that contains \( (B, n_0) \) and it is closed under the 'directly linked' relation.

In a symmetrical way one can define the above notions also for a block of \( i \)-attributes.

**Lemma 4.3.** Consider a node \( n \) of \( t' \) and let \( B \) be a block of \( s \)-attributes of \( \text{par}(n) = \Pi \). Assume that for some descendant \( m \) of \( n \) there is an attribute \( (b, m) \) such that:

(i) for some \( i \)-attribute \( (a, n) \) of \( n \) there is a path in \( D(t') \) from \( (a, n) \) to \( (b, m) \), and

(ii) there is a path in \( D(t') \) from \( (b, m) \) to an attribute of a block linked to \( (B, n) \).

Let \( (D_1, n) \) be the block containing \( (a, n) \) and \( (D_2, m) \) the block containing \( (b, m) \). The following two points hold:

(a) \( \Pi \) contains an edge \( D_1 \rightarrow B \),

(b) \( (D_2, m) \) is either linked to \( (B, n) \) or to a block \( (D_3, n) \) such that \( D_3 \) is a block of \( s \)-attributes and such that, in \( \Pi \), \( D_3 \rightarrow^+ B \).

**Proof.** Point (a) holds because, otherwise, \( \Pi \) would not contain the graph \( D(p - 0) \) \([\text{par}(n_1), \ldots, \text{par}(n_\gamma)]\) which is not possible because the fine pop-assignment \( \Psi \) is used for assigning the po-partitions to the nodes of \( t' \).

Point (b) is immediate from the definition of directly linked. \( \square \)

A flag-assignment for \( t' \) is a function \( F \) that assigns to each block \( (B, n) \), where \( n \) is a node of \( t' \) and \( B \in \text{par}(n) \), a pair \( F(B, n) = (v_m, v_i) \) of truth values: \( v_m \) is the
value of mod\((B, n)\) and \(v\), that of tr\((B, n)\). In what follows \(v_m\) and \(v\) will be denoted by \(F^m(B, n)\) and \(F^i(B, n)\), respectively.

For a valuation \(W\) of \(t'\), a block \((B, n)\), where \(B \in \text{par}(n)\), is \textit{correct} in \(W\) if all attributes of every block linked to \((B, n)\) are consistent in \(W\).

A couple \((W, F)\), where \(W\) is a valuation and \(F\) a flag-assignment for \(t'\), is called a \textit{WF-couple}. A WF-couple \((W, F)\) is \textit{correct} if for any block \((B, n)\), where \(B \in \text{par}(n)\): 

(i) \(F^m(B, n)\) is true iff for at least one attribute \(a\) of \(B\), \(W(a, n) \neq \text{VAL}_r(a, n)\),

(ii) if \(F^m(B, n)\) is true, then \(F^i(B, n)\) is also true,

(iii) if \(F^i(B, n)\) is true, then \((B, n)\) is correct in \(W\).

Given a valuation \(W\), a block \((B, n)\) is \textit{ripe} in \(W\) if the following holds: let \(\text{par}(n) = (M, \rightarrow)\); for every block \(A \in M\) such that \(A \rightarrow B\), \((A, n)\) is correct in \(W\). One also says that the block \((\phi, n)\) is ripe in \(W\) if for every block \(A \in M\), \((A, n)\) is correct in \(W\).

For a WF-couple \((W, F)\), \((t', W, F)\) represents the tree \(t'\) in which the attributes have the values indicated in \(W\) and the variables mod and tr of each node have the values specified in \(F\). From now on, it will be assumed that a triple \((t', W, F)\) is given as input to the DNC evaluator \(P\) and to similar programs. The computation of \(P\) defines a new WF-couple \((\tilde{W}, \tilde{F})\) by modifying \(W\) and \(F\).

\textbf{Lemma 4.4.} Assume Notation 4.2. Let \((W, F)\) be a correct WF-couple for \(t'\). For any node \(n\) that is not an ancestor of \(u\) (thus, it may be that \(n = u\)) the following two points hold:

(a) Let \(B\) be a block of \(s\)-attributes of \(\text{par}(n) = \Pi I\); if \((B, n)\) is ripe in \(W\), then the following is true: consider the execution of the program \(P(B, n)\),

\begin{verbatim}
begin 
  eval \Pi IB(n), 
  tr(B, n):= true
end;
\end{verbatim}

with input tree \((t', W, F)\); the WF-couple \((\tilde{W}, \tilde{F})\) defined by this execution is correct and, moreover, \(\tilde{F}^i(B, n)\) is true.

(b) If \((\phi, n)\) is ripe in \(W\), then the following is true: consider the execution of the call \(\text{eval } \Pi \phi(n, 0)\) with input \((t', W, F)\); the WF-couple \((\tilde{W}, \tilde{F})\) defined by this execution is correct and, moreover, all attributes of nodes that are descendants of \(n\) (including \(n\)) are consistent in \(\tilde{W}\).

In the proof we will use the fact, obvious from Construction 3 that when \(P\) sets a variable mod\((B, m)\) or tr\((B, m)\) to true, in the remainder of the computation of \(P\), neither this variable, nor the value of any attribute of \((B, m)\), will be modified.

The following fact is also used:

\textbf{Fact 4.5.} We use the terminology of the statement of Lemma 4.4. At each moment of the execution of \(P(B, n)\) the block \((B, n)\) is ripe in the current valuation.
Proof of Lemma 4.4. The proof of (a) is by induction on the height of the tree rooted in n. The base is simple. For the induction step assume that n is n0 in an occurrence \( \langle n0, \ldots, n_y \rangle \) of production \( p : X_0 \Rightarrow X_1 \ldots X_y \) in \( t' \) and that \( \text{par}(ni) = \Pi_i = \langle M_i, \rightarrow_i \rangle \).

Consider the section \( \text{OPT}([p, \langle \Pi_0, \ldots, \Pi_y \rangle], (B, 0)) \) of eval \( \Pi_0 B \) and let it be \( \text{Seq} = \text{INS}_1, \ldots, \text{INS}_y \), where each \( \text{INS}_i \) is an instruction package (see Construction 3). Let \( \langle W_{i+1}, F_{i+1} \rangle \) be the WF-couple defined by the execution of \( \text{INS}_1, \ldots, \text{INS}_y \).

Using a second induction on \( i \in [1, q] \) we want to show that \( \langle W_{i+1}, F_{i+1} \rangle \) is correct for each \( i \in [1, q] \). Let \( \text{INS}_i = \text{INS}(D, j) \) (recall that \( (D, nj) \) is a block owned by \( (B, n0) \)). There are two cases:

Case 1. \( \text{INS}(D, j) \) has the goal of, eventually, reevaluating a block of defined attributes of \( p \). Clearly, \( F_{i+1}'(D, nj) = \text{true} \), thus one must show that \( (D, nj) \) is correct in \( W_{i+1} \). To this end let us prove the following fact:

**Fact 4.6.** Every block \( (D_1, nh) \) such that, in \( g = D(p - 0)[\Pi_0, \ldots, \Pi_y], (D_1, h) \) is an immediate predecessor of \( (D, j) \) is correct in \( W_i \).

**Proof.** Two cases must be distinguished:

(1) \( h = 0 \): \( D_1 \) must be a block of \( i \)-attributes of \( \Pi_0 \) such that \( D_1 \rightarrow_0 B \), hence, \( (D_1, n0) \) is correct since \( (B, n0) \) is ripe in \( W \) and, by Fact 4.5, in \( W_i \) too.

(2) \( h \neq 0 \): again two cases must be distinguished:

(i) \( (D_1, nh) \) is owned by \( (B, n0) \), hence, for some \( k \in [1, q] \), \( k < i \), \( \text{INS}_k = \text{INS}(D_1, h) \), and the second induction hypothesis guarantees that \( (D_1, nh) \) is correct in \( W_i \);

(ii) \( (D_1, nh) \) is at the frontier of \( (B, n0) \), hence it is directly linked to a block \( (B', n0) \) such that \( B' \rightarrow_0 B \), and therefore, since \( (B, n0) \) is ripe in \( W \) and thus in \( W_i \), \( (B', n0) \) is correct in \( W_i \); this shows that \( (D_1, nh) \) is correct in \( W_i \) too.

**Proof of Lemma 4.4 (continued).** From Fact 4.6 it is easy to show that \( (D, nj) \) is correct in \( W_{i+1} \). Again two cases must be distinguished:

(i) \( (D, nj) \) is reevaluated in \( \text{INS}_i \): Clearly, the new values of all attributes of \( (D, nj) \) must be consistent in \( W_{i+1} \). This, together with Fact 4.6, proves that \( (D, nj) \) is correct in \( W_{i+1} \). Notice that this is the only applicable case if \( j = 0 \) and \( n0 = u \) (see Construction 3 part 4.1.2).

(ii) \( (D, nj) \) is not reevaluated in \( \text{INS}_i \): If \( F_i'(D, nj) = \text{true} \), then, by the second induction hypothesis, we are done, otherwise, for every block \( (D_1, nh) \) as in Fact 4.6, \( F_i''(D_1, nh) \) must be false and hence, the only way \( (D, nj) \) can be inconsistent in \( W_{i+1} \) is that some of its attributes are modified w.r.t. \( \text{VAL}_i' \) (here we use the fact that \( nj \neq u \! \)), but in this case, \( F_i'(D, nj) \) should be true: contradiction! This shows Case 1.

Case 2. \( \text{INS}(D, j) \) has the goal of eventually executing a recursive call. More precisely, if \( F_i'(D, nj) \) is false and if, for at least one immediate predecessor \( (D_1, h) \) of \( (D, j) \) in \( g \), \( F_i''(D_1, nh) \) is true, then the call ‘eval \( \Pi_j D(nj) \)’ is executed in \( \text{INS}_i \).
Let us first consider the case that this call is not executed. In this case $F'_{i+1}(D, nj)$ becomes true, thus it must be that $(D, nj)$ is correct in $W_{i+1}$. This is guaranteed by the assumption made on the position of $n$ w.r.t. $u$. Any attribute $(a, m)$ is inconsistent in $W_i$ only if in $D(t')$ there is a path $\pi$ from an attribute to-change of $u$ to $(a, m)$. Now, if $m$ is a descendant of $nj$, $\pi$ must traverse $nj$, i.e., $\pi$ must involve an $i$-attribute $b$ of $nj$ such that $W_i(b, nj) \neq VAL_{r_i}(b, nj)$. Let $t_i$ be the subtree rooted in $nj$. If there is a path in $D(t_i)$ from $(a, m)$ to an attribute of a block linked to $(D, nj)$, by Lemma 4.3(a), one has that, if the block containing the $i$-attribute $(b, nj)$ is $(D_2, nj)$, then $D_2 \to D$ and since $F''_i(D_2, nj)$ is true, the call ‘eval $II,\phi(nj)$’ should have been done: contradiction!

Let us now consider the case that the call is done. In this case we must show that $(D, nj)$ is ripe in $W_i$ (this would allow us to apply the first induction hypothesis). If $(D, nj)$ is not ripe in $W_i$, there must be a block $A$ of $II$ such that $A \to D$ and such that $(A, nj)$ is not correct in $W_i$. It is not possible that $(A, nj)$ is owned by $(B, n0)$ because, otherwise, there would be a $k < i$ such that $INS_k = INS(A, j)$ and thus, $F'_{i+1}(A, nj)$ would be true, contradicting, by the second induction hypothesis, the above assumption about $(A, nj)$. Then, the only possibility left is that $(A, nj)$ is directly linked to a block $(B', n0)$, where $B'$ is a block of $s$-attributes of $II_0$ such that $B' \to B$. This is not possible either because it contradicts our initial hypothesis that $n0$ is ripe in $W$. Hence $(D, nj)$ is ripe in $W_i$ and, thus, by the first induction hypothesis, $(W_{i+1}, F_{i+1})$ is correct and $(D, nj)$ is correct in $W_{i+1}$.

The proof of (b) is again by induction on the height of the tree rooted in $n$. For the call ‘eval $II,\phi(n, 0)$’ one can define the sequence of instruction packages $Seq = INS_1, \ldots, INS_n$, exactly as in the proof of (a). Recall from Construction 3 part 4.1.2 that $Seq$ consists of a prefix $INS_1, \ldots, INS_h$, $h < q$, of ‘normal’ instruction packages followed by $\gamma$ instruction packages for calling recursively (when necessary) the top-down procedures at the sons $n1, \ldots, ny$ of $n$.

Using the same argument as in the proof of (a), one can show that at the end of the execution of $INS_1, \ldots, INS_n$, the WF-couple $(W_{h+1}, F_{h+1})$ produced is correct and that every block $(B, nj)$, $j \in [0, \gamma)$, is correct in $W_{i+1}$. This implies that when each call ‘eval $II,\phi(nj, 0)$’ is performed (during the execution of $INS_{h+1}, \ldots, INS_\gamma$), $(\phi, nj)$ is ripe in the current valuation. By the induction hypothesis, this shows point (b).

Recall the role played in the proof of Lemma 4.4 by the assumption about the position of node $n$ w.r.t. $u$: It was necessary for showing that no ‘useful’ recursive call was skipped. Consider now the case that $n$ is on the path from $u$ to the root of $t'$ and that the current WF-couple $(W, F)$ is such that for each block $(D, nj)$, where $nj$ is the son of $n$ on the path from $n$ to $u$, $F'(D, nj)$ is true, i.e., $(D, nj)$ is correct in $W$. It is easy to see that, in this case, Lemma 4.4 still holds. Thus, we have the following result.

**Corollary 4.7.** Assume the terminology of Lemma 4.4, but let $n$ be a node in the situation described above. Lemma 4.4(a) and (b) are true for such an $n$ and, moreover,
Lemma 4.4(b) holds also if one substitutes the call ‘eval IIϕ(n, j)’ for that of the statement of Lemma 4.4.

Lemma 4.8. Assume Notation 4.2. Let ⟨W, F⟩ be a correct WF-couple. For any node n on the path from u to the root of t', let A be a block of i-attributes of II = par(n). If (A, n) is ripe in W, then the execution of the program,

\begin{verbatim}
begin eval II\(\alpha)(n); tr(A, n) := true end;
\end{verbatim}

with input ⟨t', W, F⟩, defines an WF-couple ⟨\(\bar{W}, \bar{F}\)⟩ that is correct and such that \(\bar{F}'(A, n)\) is true.

Proof. The proof, being very close to that of Lemma 4.4, is only sketched. The proof is organized again as a double induction: The first induction operates on the depth of the node n in t' and the second one on the number of instruction packages of the appropriate section of eval II\(\alpha\). The proof of this second induction differs from the corresponding proof of Lemma 4.4 only because one has to consider three cases, instead of two, for the block (D, j) such that INS\(\omega\) = INS(D, j): in addition to the cases that (0, j) corresponds to the evaluation of defined attributes of the production applied above n or to a recursive call at the father of n, (D, j) may correspond to a recursive call to a brother of n. This new case is handled using Lemma 4.4. □

Theorem 4.9. Construction 3 is correct, i.e., given any G, P and t' as in Notation 4.2, P computes a consistent full valuation for t'.

Proof. Notation 4.2 is used. Consider the execution of the main program of P (see Fig. 13) with input ⟨t', VAL\(\tau\), F\(\tau\)⟩, where F\(\tau\)(B, m) = (false, false) for all blocks of t'. This execution consists of a sequence of phases, where the first one takes place at node u, the second one at the father of u, and so on. The phase concerning u consists in the execution of section (2) of the main program of P and every successive phase consists in the execution of sections (3b) and (3c) of this program. Let m₁, ..., mₖ be the nodes of t' interested by this sequence of phases, that is, m₁ = u, m₂ is the father of u and so on. ⟨\(W_{i+1}, F_{i+1}\)⟩ is the WF-couple produced after the ith phase.

Let us consider the first phase at node m₁ = u. Assume, as in Fig. 13, that par(u) = II and that S = ⟨B₁, ..., Bₙ⟩ is the sequence containing all blocks of II and respecting \(\Pi\) used by P. The initial WF-couple ⟨VAL\(\tau\), F\(\tau\)⟩ is correct and B₁ is clearly ripe in VAL\(\tau\). Thus, the appropriate one of Lemmas 4.4(a) and 4.8 shows that after the execution of ‘eval II\(\phi\)(u)’ one still has a correct WF-couple ⟨\(\bar{W}, \bar{F}\)⟩. Since B₁ is correct in \(\bar{W}, (B₂, u)\) is ripe in \(\bar{W}\) and one can continue applying Lemmas 4.4(a) and 4.8. This shows that when P executes ‘eval II\(\phi\)(n, 0)’, (\(\phi, n\)) is ripe in
the current valuation and, hence, by Lemma 4.4(b), the produced WF-couple \((W_2, F_2)\) is correct and, moreover, all attributes of nodes that are descendants of \(u\) (including \(u\)) are consistent in \(W_2\).

For \(m_2\) a similar reasoning can be applied. Let \((D_1, \ldots, D_h)\) be the sequence of all blocks of \(II_2 = \text{par}(m_2)\) used by \(P\) (part (3b) of the main program): consider the call ‘eval II_2D_1(m_2)’:

(i) If \(P\) does not execute it, then \(F'_2(D_1, m_2) = \text{true}\) and, thus \((D_1, m_2)\) is correct in \(W_2\).

(ii) If \(P\) executes it then, clearly, \(D_1\) is ripe in \(W_2\) and one can apply either Corollary 4.7 or Lemma 4.8 for showing what we want.

Clearly, this reasoning can be applied for all the \(h\) calls at \(m_2\), after which \((\phi, m_2)\) is ripe and Corollary 4.7 tells us that all attributes below \(m_2\) are consistent in the defined valuation \(W_3\).

The same reasoning can be applied for all remaining nodes \(m_3, \ldots, m_k\). All attributes of \(t'\) are consistent in \(W_{k+1}\):

(i) If \(m_k\) is the root of \(t'\), this is obvious.

(ii) If \(m_k\) is not the root of \(t'\), then for none of its blocks \(F''_{k+1}\) is true. This, together with the fact that all these blocks are correct in \(W_{k+1}\) and that \(m_k\) is an ancestor of \(u\), implies immediately that all attributes of \(t'\) are consistent in \(W_{k+1}\) (no attribute of \(m_k\) is to-change). \(\square\)

**Theorem 4.10.** The DNC evaluators of Construction 3 are optimal, i.e., for any \(G\), \(P\) and \(t'\) as in Notation 4.2, for producing the consistent full valuation of \(t'\), \(P\) performs a number of operations linear in the number of attributes to-change of \(t'\).

**Proof.** The optimality of \(P\) follows from the following four points (we assume that \(t'\) contains at least one attribute to-change):

1. From Construction 3 it follows that a recursive call ‘eval II_B(n)’ of a procedure, of any of the three types, with \(n \neq u\) is executed only when some attributes of \(n\) are modified. This implies that the number of nodes of \(t'\) visited by \(P\) is linear in the number of attributes to-change in \(t'\). Let \(R\) be the portion of \(t'\) containing all the nodes visited by \(P\) (a node \(n\) is visited by \(P\) if \(P\) defines either above\((n)\) or below\((n)\)).

2. For any node \(n\) in \(R\), if \(II = \text{par}(n)\) and \(B\) is a block of attributes of \(II\), then the attributes of \((B, n)\) are reevaluated at most once by \(P\): the variable \(\text{tr}(B, n)\) is tested before evaluating and set to true when the reevaluation is performed.

3. For each block \((B, n)\), as in (2), the variables \(\text{mod}(B, n)\) and \(\text{tr}(B, n)\) are tested at most \(3\beta(\alpha + 1)\) times, where \(\alpha\) is an upper bound on the number of blocks of i-attributes and of s-attributes in the po-partitions of \(II_A\), and \(\beta\) is an upper bound on the number of tests to the same variable \(\text{tr}(B, n)\) or \(\text{mod}(B, n)\) performed in the procedures of \(P\). A rough analysis suffices for showing this fact. For simplicity, \(\text{tr}(B, n)\) and \(\text{mod}(B, n)\) are treated together in the analysis that follows and they are called the variables of \((B, n)\).
The variables of \((B, n)\) can be tested in any of the following ways:

(a) in section (3b) of the main program of \(P\) (see Fig. 13),
(b) during the execution of a procedure (of any type) called at node \(n\),
(c) during the execution of a procedure either top-down\(_1\) or top-down\(_2\) called at the father of \(n\),
(d) during the execution of a bottom-up procedure called at a son of \(n\).

Let us count how many times the variables of \((B, n)\) can be tested in each of the above four ways:

Clearly, one test only can be done in way (a).

By point (1) above, the following holds: \(\beta(\alpha+1)\) tests at most can be done in way (b): at most \(\alpha\) top-down\(_1\), \(\alpha\) bottom-up, and \(1\) top-down\(_2\) procedures can be called at \(n\). If \(B\) is a block of \(i\)-attributes, then top-down\(_1\) and top-down\(_2\) procedures can test its variables. Otherwise, if \(B\) is a block of \(s\)-attributes, the bottom-up procedures can test it. Again at most \(\beta(\alpha+1)\) tests of the variables of \((B, n)\) can be done in way (c): \(\alpha\) top-down\(_1\) and \(1\) top-down\(_2\) calls at the father of \(n\). At most \(\alpha\beta\) tests of the variables of \((B, n)\) can be done in way (d). This case applies only if \(n\) is different from the rep-node \(u\) and it is on the path from \(u\) to the root of \(t'\). Only in this case bottom-up procedures can be called at a son of \(n\) (viz., the son on the path from \(u\) to \(n\)). There are at most \(\alpha\) such calls, hence, at most \(\alpha\beta\) tests.

Summing up the four bounds we get

\[1 + 2\beta(\alpha + 1) + \alpha\beta < 3\beta(\alpha + 1).\]

(4) For each node \(n\) of \(R\), the number of tests to the variable \(\text{vis}(n)\) performed by \(P\) is bounded by \(2(\alpha + 1):\) such a test can be done in the main program of \(P\) (point (3a); see Fig. 13) and during the execution of a procedure called at \(n\). By point (1) above, there can be at most \(2\alpha + 1\) such calls (\(\alpha\) top-down\(_1\), \(1\) top-down\(_2\), and \(\alpha\) bottom-up) and each can perform at most one test. \(\Box\)

5. Conclusions

The most interesting aspects of the present work are, in our opinion, the following three:

(1) It gives a unifying graph-based characterization of ANC AGs that allows to better understand the relation among different results concerning this class.
(2) It shows how the same graph approach can be used for characterizing the DNC AGs.
(3) It shows that these characterizations are a natural basis for constructing (in an essentially similar way) classical evaluators for ANC AGs and incremental evaluators for DNC AGs.

In the future we intend to study the problem of transforming a given good pop-assignment \(IIA\) into another one \(IIA'\) in such a way that the ANC evaluator corresponding to \(IIA'\) does less recomputation than that corresponding to \(IIA\).
Intuitively, we would like to find conditions on a subset of the po-partitions of \( \Pi A \) that allow to 'improve' them, independently from the rest of \( \Pi A \). If the recomputation of the evaluator based on \( \Pi A \) is due (also) to po-partitions satisfying these conditions, then one can eliminate (reduce) it by improving them being sure that the new evaluator still has a reasonable size.

References