INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic Attribute Grammars

Didier PARIGOT, Gilles ROUSSEL, Martin JOURDAN, Étienne DURIS

N° 2881

Mai 1996

THÈME 2

Rapport de recherche

N 0249-6399

Dynamic Attribute Grammars

Didier PARIGOT, Gilles ROUSSEL, Martin JOURDAN, Étienne DURIS

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oscar

Rapport de recherche n°2881 — Mai 1996 — 29 pages

Abstract: Although Attribute Grammars were introduced long ago, their lack of expressiveness has resulted in limited use outside the domain of static language processing. With the new notion of *Dynamic Attribute Grammars* defined on top of *Grammar Couples*, informally presented in a previous paper, we show that it is possible to extend this expressiveness and to describe computations on structures that are not just trees, but also on abstractions allowing for infinite structures. The result is a language that is comparable in power to most first-order functional languages, with a distinctive declarative character.

In this paper, we give a formal definition of Dynamic Attribute Grammars and show how to construct efficient visit-sequence-based evaluators for them, using traditional, well-established AG techniques (in our case, using the FNC-2 system)[*].

The major contribution of this approach is to restore the intrinsic power of Attribute Grammars and re-emphasize the effectiveness of analysis and implementation techniques developed for them.

Key-words: Attribute grammars, static analysis, implementation, dynamic semantics, applicative programming

(Résumé : tsvp)

[*] See `http://www-rocq.inria.fr/oscar/FNC-2/` for more information.

Unité de recherche INRIA Rocquencourt

Les grammaires attribuées dynamiques

Résumé : Bien que les grammaires attribuées aient été introduites il y a longtemps, leur manque de pouvoir d'expression les a confinées dans le domaine du traitement statique des langages de programmation. Avec les nouvelles notions de *grammaires attribuées dynamiques* définies sur des *couples de grammaires*, présentées informellement dans une précédente publication, nous montrons qu'il est possible d'étendre cette expressivité et de décrire des calculs sur des structures qui ne sont pas uniquement des arbres, mais des abstractions qui rendent compte de structures infinies. Nous obtenons ainsi un langage dont le pouvoir d'expression est comparable à celui de la plupart des langages fonctionnels du premier ordre, avec un côté déclaratif beaucoup plus marqué.

Dans ce rapport, nous donnons la définition formelle des grammaires attribuées dynamiques et montrons comment construire pour elles des évaluateurs efficaces à base de séquences de visites, en utilisant des techniques traditionnelles et éprouvées (dans notre cas, en utilisant notre système FNC-2)[**].

La principale contribution de cette approche est de redonner toute leur puissance d'expression aux grammaires attribuées et de mettre en lumière l'efficacité des techniques d'analyse et d'implantation qui ont été développées pour elles.

Mots-clé : Grammaires attribuées, analyse statique, implantation, sémantique dynamique, programmation applicative

[**] Voir `http://www-rocq.inria.fr/oscar/FNC-2/` pour plus d'informations.

1 Introduction

Attribute Grammars were introduced thirty years ago by Knuth [Knu68] and, since then, they have been widely studied [DJL88, DJ90, AM91, Paa95]. An Attribute Grammar is a declarative specification that describes how attributes (variables) are computed for rules in a particular syntax (i.e., it is syntax-directed). They were originally introduced as a formalism for describing compilation applications; they were intended to describe how to decorate a tree, and could not be easily thought about in the absence of the structure (tree) representing the program to compile. In this application area, Attribute Grammars were recognized as having these two important qualities:

- they have a natural *structural decomposition* that corresponds to the syntactic structure of the language, and
- they are *declarative* in that the writer only specifies the rules used to compute attribute values, but not the order in which they will be applied.

The main question about the formalism was: "Is it possible to produce usable code from an Attribute Grammar specification?" Most research in the area has focused on the automatic production of efficient code, and very good results have been obtained, in particular, efficient implementation techniques for various subclasses of Attribute Grammars and static, global analysis methods that are generally quite precise (see [Jou92, Alb91, Paa95] for a survey and bibliography).

In spite of that, Attribute Grammar specifications are still not as widely used as they could be. Because of their historical roots in compiler construction, the notion of (physical) tree was considered as the only way to direct computations. This is the main cause for their lack of use and lack of expressiveness. Some works have attempted to respond to this problem by proposing extensions to the classical Attribute Grammar formalism, for instance Higher-Order Attribute Grammars [SV91], Circular Attribute Grammars [Far86], Multi-Attributed Grammars [Att89] or Conditional Attribute Grammars [Boy96]. The main difference between these works and ours is the methodology used to attack the problem. All of them, in a first step, propose a linguistic extension designed to make the expression of a particular application easier (for instance, data-flow analysis for Circular Attribute Grammars) and, in a second step, wondered how this extension could be implemented. In contrast, our approach [PDRJ95, PDRJ96] was, first, to precisely characterize the intrinsic power of the classical formalism and, thereafter, to derive the language extensions that allow to fully exploit this power. The basic observation is that the notion of *grammar* does *not* necessarily imply the existence of a (physical) *tree*.

In fact, our view of the grammar underlying an Attribute Grammar is similar to the grammar describing all the call trees for a given functional program or all the proof trees for a given logic program: the grammar precisely describes the various possible flows of control. In this context, a production describes an elementary recursion scheme (control flow) [CFZ82], whereas the semantic rules describe the computations associated with this scheme (data flow).

It is very important at this point to observe that all the theoretical and practical results on Attribute Grammars (in particular, the algorithms for constructing efficient evaluators) are based *only* on the abstraction of the control flow by means of a grammar and not at all on how its instances are obtained at run-time.

In consequence we present two notions which comply with this view:

- *Grammar Couples* allow to describe recursion schemes independently from any physical structure and/or to exhibit a different combination of the elements of a physical structure. A grammar couple defines an association between a dynamic grammar and a physical or concrete grammar.

- *Dynamic Attribute Grammars* (DAGs) allow attribute values to influence the flow of control by selecting alternative dynamic productions. We define the new notion of *semantic rules blocks* made of nested sets of conditional semantic rules.

These extensions eliminate the major criticism against Attribute Grammars, namely, their lack of expressiveness. They provide a programming language similar to a first-order language with a functional flavor (because of the single-assignment property) that retains the distinctive declarative character of Attribute Grammars. They have been easily implemented in Olga, the input language to our FNC-2 system [JPJ+90, JP93]. An informal, example-based comparison of Dynamic Attribute Grammars with other programming paradigms appears in [PDRJ95], together with a discussion of how this leads to fruitful applications regarding analysis and implementation techniques.

In this paper, we concentrate instead on the definition and implementation of DAGs. At this point, the semantics of DAGs is given by their functional implementation, as described here; we are working on a more elegant formulation of the semantics, which will be the subject of a forthcoming paper.

The remainder of this article is divided in two sections. The first one presents successively the classical definition of Attribute Grammars, the two new notions of *Grammar Couple* and *Dynamic Attribute Grammar* and finally the construction of a classical Attribute Grammar which has the same “behaviour” as a given DAG (the *Abstract Attribute Grammar*, or AAG, associated whith the DAG). The second section demonstrates how to use classical AG-implementation techniques (evaluator generation) to produce an efficient, visit-sequence-based evaluator for a DAG.

2 Dynamic Attribute Grammars

The goal of this section is to present and define the new notion of Dynamic Attribute Grammar and their properties. After some recalls about classical Attribute Grammars, we will deal successively with Grammar Couples, Dynamic Attribute Grammars and finally Abstract Attribute Grammars.

2.1 Classical Attribute Grammars

In this subsection we recall the classical Attribute Grammars definitions, and give some notations useful for the sequel.

Definition 2.1 (Context-Free Grammar) *A* context-free grammar *is a tuple* $G = (N, T, Z, P)$ *in which:*

- N *is a set of non-terminals;*
- T *is a set of terminals,* $N \cap T = \emptyset$*;*
- Z *is the root non-terminal (start symbol),* $Z \in N$*;*

- *P is a set of productions,*
  $p : X_0 \rightarrow X_1 \ldots X_n$ *with* $X_0 \in N$ *and* $X_i \in (T \cup N)$.

In this paper, we will forget about terminals and parsing problems and consider a grammar as an algebraic definition of a family of trees (or terms or structures).

Definition 2.2 (Attribute Grammar) *An* Attribute Grammar *is a tuple* $AG = (G, A, F)$ *where:*

- $G = (N, T, Z, P)$ *is a context-free grammar as in definition 2.1;*
- $A = \bigcup A(X)$ *is a set of attributes attached to* $X \in N$, $A(X) = H(X) \uplus S(X)$ *with* $H(X)$ *the* inherited *attributes of* X *and* $S(X)$ *the* synthesized *ones;*[1]
- $F = \bigcup F(p)$ *is a set of semantic rules, where* f_{p,a,X_i} *designates the semantic rule defining the attribute occurrence* $a(X_i)$ *in production* $p : X_0 \rightarrow X_1 \ldots X_n$ *and* $a \in A(X_i)$.

In the previous definitions, there is some ambiguity in the use of symbol X_i. In the CFG definition they represent non-terminals meanwhile in the AG definition they represent both the non-terminal occurrence (labeled by its position in the production) and the non-terminal (type) itself. However, the position of a name in a production is only relevant for X_0, or to distinguish two non-terminal occurrences and their types. Therefore, we consider a production as a set of distinct names (with a specific one for the left-hand side), each with a type. So we now give a more precise definition of a production.

Definition 2.3 (Production) *Let* $\mathcal{V}$ *be a universal finite set of* names. *A production* $p : X_0 \rightarrow X_1 \ldots X_n$ *in a CFG is a tuple* $((X_0, \mathcal{V}_p), type_p)$ *in which:*

i. $\mathcal{V}_p = \{X_1, X_2, \ldots, X_n\} \subset \mathcal{V}$, *with* $n = Card(\mathcal{V}_p)$, *and* $X_0 \in \mathcal{V} - \mathcal{V}_p$;

ii. $type_p : \mathcal{V}_p^+ \rightarrow N \cup T$, *where* $\mathcal{V}_p^+ = \{X_0\} \cup \mathcal{V}_p$, *is a function which associates to each name a unique type in the set of non-terminals and terminals.*

The first condition means that all the names appearing in the right hand side of the production are distinct and that their order is not relevant.

In the sequel of this paper, we will use the clearest of our two notations for a production—$p : X_0 \rightarrow X_1 \ldots X_n$ or $((X_0, \mathcal{V}_p), type_p)$—according to the context.

We now give some notations relative to such a production:

- $LHS(p) = X_0$ and $RHS(p) = \mathcal{V}_p$.
- $W_u(p) = \{a(X_0) \mid a \in H(type_p(X_0))\} \cup \{a(X) \mid X \in \mathcal{V}_p, a \in S(type_p(X))\}$ is the set of *input* or *used* attribute occurrences in p.
- $W_d(p) = \{a(X_0) \mid a \in S(type_p(X_0))\} \cup \{a(X) \mid X \in \mathcal{V}_p, a \in H(type_p(X))\}$ is the set of *output* or *defined* attribute occurrences in p.
- $W(p) = W_u(p) \cup W_d(p)$ is the set of all *attribute occurrences* in p.

[1] $\uplus$ denotes the disjoint union (partition).

- $D(p) = (W(p), E(p))$ is the *dependence graph* for production p, with $E(p) = \{a(X) \leftarrow b(Y) \mid b(Y)$ is an argument of rule $f_{p,a,X}\}$.

We will deal only with well-formed AGs, so $F(p)$ shall contain exactly one semantic rule defining each *output* occurrence. Furthermore, all our AGs will be in normal form, which means that all attribute occurrences in the right-hand side of a semantic rule or in a condition (see below) must be *input* occurrences.

2.2 Dynamic Attribute Grammars

As said in the introduction, the basis for a Dynamic Attribute Grammar is a grammar which describes the control flow (recursion scheme) of the intended application. This control flow can depend purely on attribute values but also on the shape of some physical tree, which will then be a distinguished parameter to the evaluator. Hence we have to make a difference, but also establish a correspondence, between the grammar which describes the concrete structure and the one which describes the computation scheme (which will "contain" the former, in some sense). This is the motivation for the notion of Grammar Couple defined below. Grammar couples are similar to, although more general than, *grammar interpretations* as defined by Filé [Fil83].

Definition 2.4 (Grammar Couple) *A* Grammar Couple $G = (G_d, G_c, \mathcal{C}oncrete)$ *is a pair of context-free grammars* $G_d = (N_d, T_d, Z_d, P_d)$ *and* $G_c = (N_c, T_c, Z_c, P_c)$*, as in definition 2.1, and a function* $\mathcal{C}oncrete : P_d \times \mathcal{V} \to (P_c \times \mathcal{V}) \cup \perp$*, where:*

1. $N_c \subseteq N_d$; $T_d = T_c$; *if* $N_c \neq \emptyset$ *then* $Z_d = Z_c$.

2. $\forall p_d \in P_d$*, we have:*

   i. $\forall X \in \mathcal{V}^+_{p_d}$, $type_{p_d}(X) \in (N_d - N_c) \Rightarrow \mathcal{C}oncrete(p_d, X) = \perp$;

   ii. $type_{p_d}(LHS(p_d)) \in (N_d - N_c) \Rightarrow \forall X \in RHS(p_d)$, $type_{p_d}(X) \in (N_d - N_c)$;

   iii. $type_{p_d}(LHS(p_d)) \in N_c \Rightarrow \exists!\ p_c \in P_c$ *such that:*

      – $\mathcal{C}oncrete(p_d, LHS(p_d)) = (p_c, LHS(p_c))$ *and*
      $type_{p_d}(LHS(p_d)) = type_{p_c}(LHS(p_c))$;

      – $\forall X \in RHS(p_d)$, $type_{p_d}(X) \in N_c \Rightarrow \exists Y \in \mathcal{V}^+_{p_c}$ *such that*
      $\mathcal{C}oncrete(p_d, X) = (p_c, Y)$ *and* $type_{p_d}(X) = type_{p_c}(Y)$.

3. $\forall p, q \in P_d$ *such that* $type_p(LHS(p)) = type_q(LHS(q))$ *and*
   $\mathcal{C}oncrete(p, LHS(p)) = \mathcal{C}oncrete(q, LHS(q))$*, we have:*

   i. $LHS(p) = LHS(q)$;

   ii. $\forall X \in \mathcal{V}_p \cap \mathcal{V}_q, type_p(X) = type_q(X)$;

   iii. $\forall X \in \mathcal{V}_p \cap \mathcal{V}_q, \mathcal{C}oncrete(p, X) = \mathcal{C}oncrete(q, X)$.

Given the above constraints, we can unambiguously extend the function $\mathcal{C}oncrete$ to productions p_d of P_d as follows:

$$\mathcal{C}oncrete(p_d) = \begin{cases} p_c & \text{if } \mathcal{C}oncrete(p_d, LHS(p_d)) = (p_c, LHS(p_c)) \\ \perp & \text{if } \mathcal{C}oncrete(p_d, LHS(p_d)) = \perp \end{cases}$$

Concrete production $p \in P_c$:

p: `while:STAT -> cond:COND body:STAT`

Dynamic productions p_r and $p_t \in P_d$:

p_r: `w=while:STAT -> cond=cond:COND body=body:STAT w-rec=while:STAT`

p_t: `w=while:STAT -> cond=cond:COND`

Figure 1: Part of a grammar couple for the **while** statement

In the previous definition, G_d and G_c respectively represent the *dynamic* and *concrete* grammars, and $\mathcal{C}oncrete$ gives the concrete production (or name) corresponding to a dynamic one, i.e. a physical tree (or node). When the value of this function is $\perp$ (undefined), it means that the argument is a purely dynamic, or "abstract" object (it corresponds to some pure recursion scheme). More precisely:

- Condition 2.*ii*. means that a pure dynamic object (non-terminal) may not yield any concrete one.
- Condition 2.*iii*. means that, in a dynamic production p_d, if the LHS type is concrete, then there exists a unique corresponding concrete production p_c, which has the same type as LHS. Furthermore, for all non-terminals with a concrete type in the RHS of p_d, there exists in p_c a corresponding non-terminal with the same type. In other words, each physical structure yielded by p_d exists in p_c (possibly among others purely dynamic objects).

  Note that a given physical structure may be referenced more than once in the dynamic production and that the concrete LHS, which by definition is associated with the dynamic LHS, may also be referenced again in the dynamic RHS. *These "special effects" are the essence of DAGs* and allow to express computations that were deemed impossible with classical AGs. The latter effect is illustrated in our **while** example (see below), whereas the former is used in the *double* example of [PDRJ95].
- Condition 3 stems from the constraint that, for two productions with the same LHS type and the same associated $\mathcal{C}oncrete$[2], the LHS must have the same name and all names common to both productions must have the same type. This implies in particular that, if the corresponding $\mathcal{C}oncrete$ counterpart of a such common name is not undefined, it is actually the same concrete object.

As an example, let's see how to describe the structure and dynamic semantics of the **while** statement as a grammar couple $G = (G_d, G_c, \mathcal{C}oncrete)$. If `STAT,COND` $\in N_d \cup N_c$ respectively represent statements and boolean conditions, figure 1 shows the productions for the **while** statement. In this example, `name:TYPE` means that `TYPE` is the type of `name` and `name_d=name_c` means that $\mathcal{C}oncrete(p_d, \texttt{name\_d}) = (p_c, \texttt{name\_c})$.[3] $p \in P_c$ is the concrete production which describes that a **while** statement is made of a condition and a body statement. p_r and $p_t \in P_d$ are two dynamic productions which respectively represent the recursive behaviour of a **while** structure (p_r) as long as the condition is true and the termination case (p_t) when the condition becomes false.

We now introduce the notion of semantic rules block, as a conditional structure (decision tree) for semantic rules and productions.

[2] with possibly $\mathcal{C}oncrete = \perp$.

[3] Where p_d and p_c are unambiguously defined by the context.

```
⟨ h.env(cond) := h.env(w),                          — common semantic rule R
  ⟨ (s.c(cond)),                                    — boolean expression
    ⟨ w=while:STAT -> cond=cond:COND body=body:STAT w-rec=while:STAT,
        h.env(body) := h.env(w)                     — true case :
        h.env(w-rec) := s.env(body)                 — ⟨p_r, R'⟩
        s.env(w) := s.env(w-rec) ⟩,
    ⟨ w=while:STAT -> cond=cond:COND,               — false case :
        s.env(w) := h.env(w) ⟩ ⟩ ⟩                  — ⟨p_t, R''⟩
```

Figure 2: The semantic rules block for the **while** statement

Definition 2.5 (Semantic Rules Block) *A semantic rules block b is inductively defined as follows:*

$$b = \langle R, \langle e, b, b\rangle\rangle \mid \langle p, R\rangle$$

where R is a possibly empty set of (unconditional) semantic rules, e is a condition (boolean expression over attribute occurrences) and p is a production.

Figure 2 presents the semantic rules block describing the denotational-like semantics of our running example of the **while** statement. Attributes names are prefixed by $h.$ for inherited, and $s.$ for synthesized. The attribute env represents the execution environnement (store, etc.) of a statement and $s.c$ carries the value of the boolean condition.

In a block, semantic rules are associated with any node of the decision tree whereas the productions appear only at the leaves. The following definition shows how a block is "flattened" into a collection of traditional productions-with-semantic-rules.

Definition 2.6 ($\mathcal{R}^b$ set) *For each block b, $\mathcal{R}^b$ is the set of all semantic rules in b, qualified by the conjunction (path) of conditions that constrain (enable) them and the production to which they are attached:*

- $\mathcal{R}^{\langle p,R\rangle} = \{((\varepsilon, p), R)\}$
- $\mathcal{R}^{\langle R,\langle e,b_{true},b_{false}\rangle\rangle} =$
  $$\begin{array}{ll} let & \mathcal{R}^{b_{true}} = \cup_i((c_i,p_i),R_i), \\ & \mathcal{R}^{b_{false}} = \cup_j((c_j,p_j),R_j) \\ in & \cup_i(((e,true).c_i,p_i),R\cup R_i) \bigcup \cup_j(((e,false).c_j,p_j),R\cup R_j). \end{array}$$

Figure 3 illustrates the previous definition on our example.

For a given semantic rules block b, we define $\mathcal{PR}^b$ as the set of all productions in b: $\mathcal{PR}^b = \{p \mid ((c,p),R) \in \mathcal{R}^b\}$. We say that the pair $((c,p),R)$ is *well-formed* if the semantic rules set R is well-formed for the production p and each condition e in path c refers only to input attribute occurrences of p.

We are now ready to define complete Dynamic Attribute Grammars.

Definition 2.7 (Dynamic Attribute Grammar) *A* Dynamic Attribute Grammar *is a tuple $AG = (G, A, F)$ where:*

- *$G = (G_d, G_c, \mathcal{C}oncrete)$ is a grammar couple as in definition 2.4;*
- *$A = \bigcup_{X\in N_d} A(X)$ is a set of attributes;*

```
{(((s.c(cond), true),
        w=while:STAT -> cond=cond:COND body=body:STAT w-rec=while:STAT),
            h.env(cond) := h.env(w)
            h.env(body) := h.env(w)
            h.env(w-rec) := s.env(body)
            s.env(w) := s.env(w-rec)),
(((s.c(cond), false), w=while:STAT -> cond=cond:COND),
            h.env(cond) := h.env(w)
            s.env(w) := h.env(w))}
```

Figure 3: The $\mathcal{R}^b$ set for the **while** semantic rules block

- *F is a set of semantic rules blocks such that:*
  1. *$\forall b \in F$, every $((c,p),R) \in \mathcal{R}^b$ is well-formed;*
  2. *$\forall p \in P_d, \exists!\ b \in F$ such that $p \in \mathcal{PR}^b$;*
  3. *$\forall p,q \in P_d$, with $p \in \mathcal{PR}^{b_i}$ and $q \in \mathcal{PR}^{b_j}$, such that $type_p(LHS(p)) = type_q(LHS(q)) = X$, we have:*
     - $X \in (N_d - N_c) \Rightarrow b_i = b_j$;
     - $X \in N_c \Rightarrow (b_i = b_j \Leftrightarrow \mathcal{C}oncrete(p) = \mathcal{C}oncrete(q))$.

Given these definitions, we can make the following remarks:

- Condition 2 above means that a given production p can appear more than once in a given semantic rules block, but it can not appear in more than one semantic rules block.
- Condition 3 means that for a production p:
  - if the (type of the) LHS of p is not a concrete non-terminal, then the block in which this production appears contains *all* the productions defining this (abstract) non-terminal;
  - if it is a concrete one, then its block contains all and only the productions that are associated with the same concrete production as p.

  Hence it is obvious to extend the function $\mathcal{C}oncrete$ to blocks.
- There exists no condition relating names from different blocks, so the scope of a name is effectively a block.
- Condition 3 of definition 2.4 means that, for any two productions in a same block (same type and $\mathcal{C}oncrete$ associated with LHS), their LHS must have the same name, and all names common to both productions must have the same type too. So it makes sense to define the function $type_b$ for a block b: $\forall p \in \mathcal{PR}^b, \forall X \in \mathcal{V}_p^+, type_b(X) = type_p(X)$.

Another remark is that it is not forbidden for the start symbol of the dynamic grammar to have inherited attributes. In fact, what a Dynamic Attribute Grammar describes is a function taking as arguments:

- values for all the inherited attributes of the start symbol, and
- if the concrete grammar in the grammar couple is not empty, a concrete tree described by this grammar,

and which returns the values of the synthesized attributes of the start symbol. The computation of the attributes is defined in an "obvious" way and is guided at each "dynamic node" by the values of the various conditions and, when relevant, by the production applied at the corresponding concrete node. The formal definition of the semantics of a DAG is the topic of our present work and will be given in a forthcoming paper; in the meantime, it will be defined by its implementation, as described below, and we hope that the sequel of this paper and the examples in [PDRJ95] will help the reader intuitively grasp the semantics and operation of a DAG.

2.3 Abstract Attribute Grammars

We claimed in [PDRJ95] that Dynamic Attribute Grammars could be implemented using the same techniques as classical AGs. The basic idea, called *plane shift* in [Jou92] and indeed used in FNC-2 to implement DAGs, is simple:

1. build from the given DAG a classical AG which has the same "behaviour" (syntax—i.e., recursion scheme—and dependencies—i.e., data flow);
2. generate the evaluator for this classical AG;
3. "patch" this evaluator so that it becomes a correct implementation of the original DAG.

In this section, we show how to construct this equivalent classical AG, which we call the *Abstract Attribute Grammar* associated with the DAG.

Let $b = \langle R, \langle e, \langle p_T, R_T \rangle, \langle p_F, R_F \rangle \rangle \rangle$ be the simplest form of a (conditional) block. Basically, the productions and semantic rules in the AAG which will reproduce the behaviour of this block are, on one hand, p_T associated with the rules in $R \cup R_T$ and, on the other hand, $\langle p_F, R \cup R_F \rangle$. This is indeed correct from the point of view of the recursion schemes and data flows, and the well-formedness conditions on the DAG will ensure that the resulting AAG will also be well-formed. The definition below formalizes this intuition and adds the very important constraint that no attribute defined by a rule in the groups subject to the condition (R_T and R_F) can be evaluated before the condition.

Definition 2.8 (Abstract Attribute Grammar) *The* Abstract Attribute Grammar *for a given Dynamic Attribute Grammar* $DAG = (G = (G_d, G_c, \mathcal{C}oncrete), A, F)$ *is a tuple* $AAG = (G_a, A_a, F_a)$ *where:*

- $G_a = (N_a, T_a, Z_a, P_a)$; $N_a = N_d$; $Z_a = Z_d$; $T_a = T_d$; $A_a = A$;
- $P_a = \{c.p_d : X_0 \to X_1 \ldots X_{n_{p_d}} \mid \exists b \in F, ((c, p_d), R) \in \mathcal{R}^b$ *with* $p_d : X_0 \to X_1 \ldots X_{n_{p_d}} \in P_d\}$;
- $F_a = \cup_{p \in P_a} F_a(p)$ *is a set of semantic rules, with* $F_a(p) = R$ *such that* $p = c.p_d$ *and* $\exists b \in F, ((c, p_d), R) \in \mathcal{F}^b$, *with* $\mathcal{F}^b$ *defined below.*

```
(s.c(cond), true).p_r : w=while:STAT ->
                          cond=cond:COND body=body:EXP w-rec=while:STAT
(s.c(cond), false).p_t : w=while:STAT -> cond=cond:COND
```

Figure 4: Productions of the Abstract AG for the **while** statement

```
{(((s.c(cond), true),
    w=while:STAT -> cond=cond:COND body=body:STAT w-rec=while:STAT),
        h.env(cond) := h.env(w)
        h.env(body) := dp(h.env(w),s.c(cond))
        h.env(w-rec) := dp(s.env(body),s.c(cond))
        s.env(w) := dp(s.env(w-rec),s.c(cond))),
(((s.c(cond), false), w=while:STAT -> cond=cond:COND),
        h.env(cond) := h.env(w)
        s.env(w) := dp(h.env(w),c(cond)))}
```

Figure 5: Modified semantic rules in the AAG for the **while** statement

In the previous definition, $c.p_d$ is just a name for a production in AAG which encodes its origins in DAG: the production (scheme) p_d, which is subject to the sequence of guards c. For example, the two productions in the AAG in figure 4, which corresponds to our example of the **while** statement, are the same as in figure 1 except that the production names are prefixed by the condition which constrain them, as in figure 3.

Let DP be the transformation which, to a given semantic rule of the form $f_{p,a,X} : a(X) := exp$ and a condition e seen as an expression over some attribute occurrences, associates the modified semantic rule $DP(f_{p,a,X}, e) : a(X) := dp(exp, e)$, where dp is the polymorphic function defined as $dp(x, y) = x$. The definition of DP extends to set of semantic rules: $DP(R, e) = \{DP(f_{p,a,X}, e) \mid f_{p,a,X} \in R\}$. The purpose of DP is to make sure that a given attribute cannot be evaluated before condition e, without altering its value.

Definition 2.9 ($\mathcal{F}^b$) *For each block b, $\mathcal{F}^b$ is the set of all semantic rules in b, qualified* and modified *by the conjunction (path) of conditions that constrain (enable) them and attached to their respective production:*

- $\mathcal{F}^{\langle p,R\rangle} = \{((\varepsilon, p), R)\}$
- $\mathcal{F}^{\langle R,\langle e,b_{true},b_{false}\rangle\rangle} =$

$$\begin{array}{ll} let & \mathcal{F}^{b_{true}} = \cup_i((c_i, p_i), R_i), \\ & \mathcal{F}^{b_{false}} = \cup_j((c_j, p_j), R_j) \\ in & \cup_i(((e, true).c_i, p_i), R \cup DP(R_i, e)) \bigcup \cup_j(((e, false).c_j, p_j), R \cup DP(R_j, e)). \end{array}$$

Figure 5 presents the productions and modified semantic rules in the AAG for the **while** statement.

To summarize, for each block of rules in the DAG, the AAG will contain the production associated with each leaf in the decision tree of the block, with the set of semantic rules that appear on the path to the root of this tree, modified to make each “subtree” of rules dependent on the condition labeling the node.

It is clear that, given an "abstract" tree that represents the same recursion scheme as some computation described by the DAG, the AAG describes the same computation over this tree: the values of the attributes will be the same and, *a posteriori*, we can check that the conditions will have the same values, too. The other additions to the AAG are pure dependencies which ensure that the evaluation of the conditions and of the attributes alternate in the "right" order. This point will be further discussed below.

At this point, we can notice that there exist no explicit dependence in the AAG which account for the "control dependence" between the various conditional expressions. It is quite possible that the outermost condition be the most contrained (in term of data dependencies).

Definition 2.10 (Evaluation classes) *A Dynamic Attribute Grammar is said to belong to some class (e.g. non-circular, l-ordered, one-pass, etc.) iff its associated Abstract Attribute Grammar belongs to this class.*

3 Visit-Sequence-Based Implementation

3.1 Introduction

As pointed out in the introduction, our extensions to the Attribute Grammar formalism can easily be added to an existing system based on static-order evaluation methods, such as FNC-2, without major modifications to the evaluator generator. This is a very attractive result, because the evaluator generator is the most intricate and most important part of an Attribute Grammar system. However, as of today, the only evaluation method for which we have studied in detail the feasibility and correctness of the "plane shift" technique described above is the visit sequence paradigm [Kas80, Eng84, Kas91], applicable to l-ordered AGs and, with a very slight extension, to strongly non-circular AGs [Par88]. Other evaluation methods will be studied in a forthcoming paper.

The visit sequence is our preferred method because: these evaluators reach the best compromise between the time and space efficiency and the generality of the AG class they can implement; this is the paradigm we have implemented in FNC-2 [JPJ+90] (for the reason just mentioned and for their versatility); and they are the easiest to transform into functions or procedures, which gives a base for our studies on the relationships between AGs and functional programming [PDRJ95].

The basic idea is as follows:

1. We construct the abstract AG corresponding to the given dynamic AG and test or make sure that it is l-ordered by exhibiting or constructing appropriate totally-ordered partitions (TOPs) of the attributes of each non-terminal.

2. Using these TOPs, we generate a separate visit sequence for each of the productions of the AAG.

3. Each of the latter productions corresponds to some "guarded production" $c.p$ in the dynamic AG. So we reintroduce in each "abstract" visit sequence marks for the evaluation of the various conditions (guards) of the dynamic production it corresponds to. For each condition, this is done as early as possible, i.e. as soon as the value of all the attribute occurrences appearing in the condition are available.

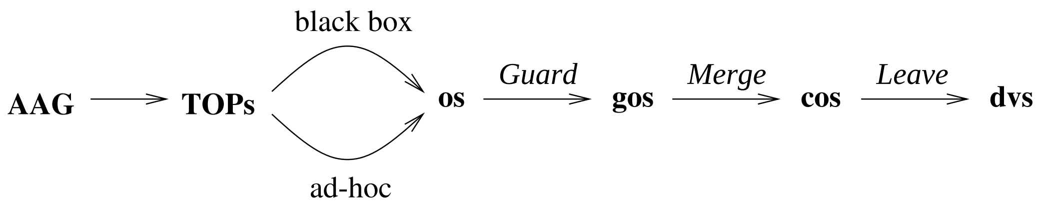

Figure 6: The basic idea

4. We then merge all the visit sequences corresponding to the same block, so as to obtain a single conditional visit sequence structured just like the decision tree of the block. To make this possible, we have to make sure that these visit sequences are "compatible", i.e. that, for a simple block of the form $\langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, the part of the guarded visit sequences for p_T and p_F that appear before the evaluation of the condition e both compute exactly the same collection of attribute occurrences. We propose two approaches to achieve this constraint.

5. We cut this tree of visit sequences in "slices" corresponding to the various visits to the LHS node, so as to make each slice a separate function, and we reintroduce at the beginning of each such function the branching code executed in previous visits.

Figure 6 illustrates this process and introduces the abbreviated names of the various objects it manipulates; these objects will be defined as needed.

The rest of this section is organized as follows. First, we present a couple of definitions and the "preprocessing" phase which produces the TOPs, together with the classical construction which produces visit sequences (more precisely, in our case, *ordered sequences*) from the dependency graphs augmented with these TOPs. Then, we present two approaches to prove that the ordered sequences we construct are compatible and can be merged. Finally, in the "post-processing" section, we show how to produce complete evaluators from the ordered sequences, as outlined above.

3.1.1 Definitions and preprocessing

The definitions of l-ordered Attribute Grammars, TOPs, augmented dependency graphs and visit sequences, together with the construction of the latter from the formers, are quite classical [Eng84, Alb91], but we repeat them here for the sake of completeness and because we introduce *ordered sequences* which are easier to reason about than visit sequences.

Definition 3.1 (l-ordered Attribute Grammars *et al.*) *Let $AG = (G, A, F)$ be a (classical) Attribute Grammar, with $G = (N, T, Z, P)$.*

- *For $X \in N$, a* Totally-Ordered Partition (TOP) *on X, T_X, is an ordered sequence $H_1(X)$, $S_1(X)$, $H_2(X)$, $S_2(X)$, $\ldots$, $H_{n_X}(X)$, $S_{n_X}(X)$, where n_X is the* number of visits *to X, such that $H(X) = \biguplus_{1\le i\le n_X} H_i(X)$, $S(X) = \biguplus_{1\le i\le n_X} S_i(X)$, $\forall i, 1 < i \le n_X, H_i(X) \ne \emptyset$ and $\forall i, 1 \le i < n_X, S_i(X) \ne \emptyset$.*

- *Given a production $p : X_0 \to X_1 \ldots X_n \in P$ and a family of TOPs $T_{X_i}, 0 \le i \le n$, the* augmented dependence graph *$\gamma(p) = D(p)[T_{X_0}, T_{X_1}, \ldots, T_{X_n}]$ is defined as follows: $\gamma(p) = (W(p), E_\gamma(p))$ where $E_\gamma(p) = E(p) \cup \{a(X_i) \to b(X_i) \mid 0 \le i \le n, \exists j, 1 \le j \le n_{X_i}, (a \in H_j(X_i) \wedge b \in S_j(X_i)) \bigvee (a \in S_j(X_i) \wedge b \in H_{j+1}(X_i))\}$.*

- *The Attribute Grammar AG is l-ordered iff there exists a family of TOPs* $\{T_X \mid X \in N\}$ *such that* $\forall p \in P$, $\gamma(p)$ *is acyclic.*

The problem of finding a family of TOPs such that a given Attribute Grammar is l-ordered for this family is well-known to be NP-complete, but various polynomial, approximate algorithms have been published [KW76, Kas80, Far83, Par88].

In the rest of this section, we'll consider only l-ordered Dynamic and Abstract Attribute Grammars, for which the family of TOPs $\{T_X \mid X \in N\}$ is given by some not further specified method.

Definition 3.2

- *Given* $a(X) \in W(p)$, $\mathcal{DD}(a(X)) = \{b(Y) \in W(p) \mid a(X) \leftarrow b(Y) \in \gamma(p)\}$. *Where necessary, we will qualify* $\mathcal{DD}$ *with the name of the production.*

- $\mathcal{DD}^+$ *is the transitive closure of* $\mathcal{DD}$. *Since* $\gamma(p)$ *is acyclic, we know that* $\mathcal{DD}^+(a(X))$, *the* cone of dependence *of* $a(X)$, *i.e. the set of attribute occurrences on which it transitively depends, is also an acyclic subgraph of* $\gamma(p)$, *with a* frontier $\mathcal{DD}^\infty(a(X)) = \{u \in \mathcal{DD}^+(a(X)) \mid \mathcal{DD}(u) = \emptyset\}$ *such that any path from* $(a(X))$ *to an element of the frontier is bounded in length.*

  We extend this notation to sets of attribute occurrences and to expressions over attribute occurrences.

- *An* ordered sequence *os on p is an ordered subset of* $W(p)$ *such that the total order on os respects the partial order* $\gamma(p)$.

- *An ordered sequence os on p is* complete *if all* $a(X) \in W(p)$ *appear in os. A complete ordered sequence is hence a valid evaluation order of all the attribute occurrences in p.*

- *For a given production p, the set of* evaluable attributes *after os is the set* $\mathcal{E}(os) = \{a(X) \in W(p) \mid \mathcal{DD}(a(X)) \subset os, a(X) \notin os\}$.

- *The function* $\mathcal{P}ick(\mathcal{E}(os)) = a(X), a(X) \in \mathcal{E}(os)$, *is a choice function.*

The notion of (complete) ordered sequence will be the basis of the proofs that we can construct compatible visit sequences. There is a mapping from ordered sequences to visit sequences and back, to be presented in the "post-processing" section, which preserves the fact that they respect the order in $\gamma(p)$.

The construction of a complete ordered sequence from an augmented dependency graph is simply a topological sort, such as performed by the algorithm in figure 7, worded with our notations.

In our journey from a Dynamic Attribute Grammar to its evaluator, we have reached a point where we can construct a classical complete ordered sequence for each production of the Abstract Attribute Grammar. These sequences have no notion of conditions. Reintroducing the latter in the former will lead to *guarded ordered sequences.*

Definition 3.3 (Guarded Ordered Sequence) *Given a complete ordered sequence os on* $p = c.p_d \in P_a$, *we define the* guarded ordered sequence $gos = \mathcal{G}uard(p, os))$ *as the result of the* $\mathcal{G}uard$ *transformation, which adds marks for the evaluation of conditions* ($cond_e$) *in the ordered sequence, in the same order as in c, and is inductively defined as follows:*

$os \leftarrow \epsilon$;
repeat
  compute $\mathcal{E}(os)$;
  $a(X) \leftarrow \mathcal{P}ick(\mathcal{E}(os))$;
  $os \leftarrow os.a(X)$;
until os is complete

Figure 7: Topological sort of $\gamma(p)$

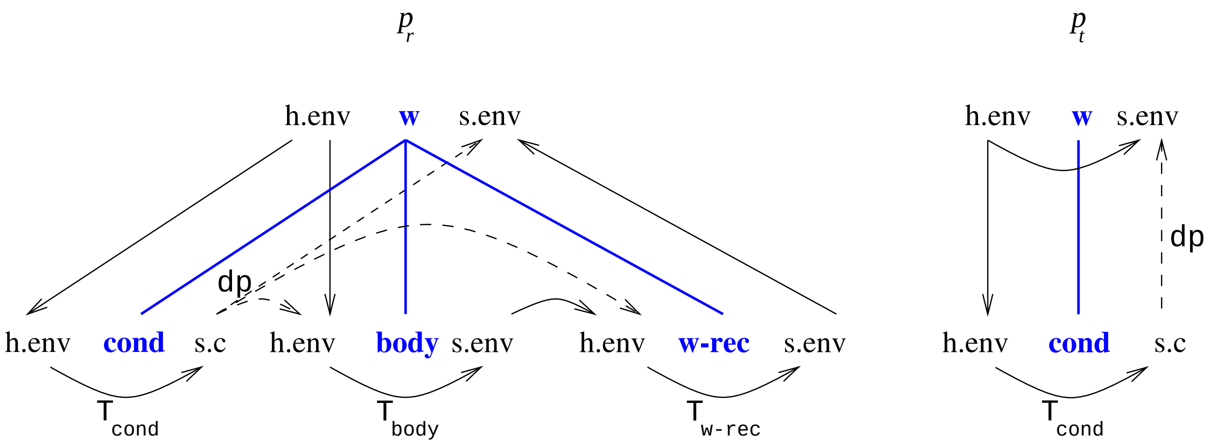

Figure 8: The dependencies of the **while** productions

$\mathcal{G}uard((s.c(cond), true).p_r, os_{p_r}) =$
  `h.env(w), h.env(cond), s.c(cond),` $cond_{s.c(cond)}$,
  `h.env(body), s.env(body), h.env(w-rec), s.env(w-rec), s.env(w)`
$\mathcal{G}uard((s.c(cond), false).p_t, os_{p_t}) =$
  `h.env(w), h.env(cond), s.c(cond),` $cond_{s.c(cond)}$, `s.env(w)`

Figure 9: Guarded ordered sequences for the **while** productions

- $\mathcal{G}uard(\varepsilon.p_d, os) = os$;
- $\mathcal{G}uard((e, v).p_e, os) = os'.cond_e.\mathcal{G}uard(p_e, os'')$, *where* $os = os'.os''$ *with* os' *such that* $\mathcal{DD}(e) \subset os'$ *and* $\forall os_1 \neq os', os' = os_1.os_2, \mathcal{DD}(e) \not\subset os_1$.

The definition of $\mathcal{G}uard$ ensures that each condition is evaluated as soon as possible, i.e. just after the evaluation of the last attribute it depends on.

In the modified semantic rules of our **while** example, presented in figure 5, the polymorphic function dp adds new dependencies in the dependence graph of each production. Figure 8 shows the dependence graphs corresponding to $(s.c(cond), true).p_r$ and $(s.c(cond), false).p_t$, with new dp dependences as dashed lines. According to these graphs, we have two associated ordered sequences, say, os_{p_t} and os_{p_r}. So, the $\mathcal{G}uard$ transformation introduce in each of them a mark, $cond_{s.c(cond)}$, and leads to the guarded ordered sequences presented in figure 9.

Now, consider a simple conditional block $b = \langle R, \langle e, \langle p_T, R_T\rangle, \langle p_F, R_F\rangle\rangle\rangle$ and the two guarded ordered sequences $gos_T = \mathcal{G}uard(p_T, os_T) = os'_T.cond_e.os''_T$ and $gos_F = \mathcal{G}uard(p_F, os_F) =$

$os'_F.cond_e.os''_F$ which will be constructed for p_T and p_F. We want to produce a *conditional ordered sequence* which will:

1. evaluate the attributes "before" the condition;
2. evaluate the condition;
3. according to the value of the latter, continue with one of the sequences or the other.

To make this possible, we have to make sure that os'_T and os'_F are *compatible*, i.e. they contain exactly the same set of attribute occurrences.

In the next sections, we present two approaches to the construction of ordered sequences and the proof that they lead to compatible ordered sequences: the first one uses the classical construction of ordered sequences but requires that we start with a more rigid AAG than the one presented earlier; the second one starts with the standard AAG but requires that the construction of ordered sequences is aware of the conditions. Then, we proceed with the construction of complete evaluators.

3.1.2 The black-box approach

Here is the fundamental theorem that we want to prove:

Theorem 3.1 *Given a block $b = \langle R, \langle e, (p_1, R_T), (p_2, R_F)\rangle\rangle$ which induces:*

- $p_T = (e, true).p_1, p_F = (e, false).p_2 \in P_a$,
- *os_T and os_F the ordered sequences generated by the topological sort algorithm,*
- *$gos_T = \mathcal{G}uard(p_T, os_T) = os'_T.cond_e.os''_T$ and $gos_F = \mathcal{G}uard(p_F, os_F) = os'_F.cond_e.os''_F$ the corresponding guarded ordered sequences,*

if $\mathcal{P}ick$ is a deterministic function, then $os'_T = os'_F$.

It means that, if, to generate the individual ordered sequences, we use an *unmodified*[4] traditional generator, provided—and this condition will easily be met in practice—that this generator is deterministic, then these sequences will be compatible. We actually have the stronger result that the subsequences between two conditions will be *identical*, rather than simply compatible.

Unfortunately, with the standard AAG defined above, it is impossible to prove this theorem. Indeed, we may encounter the following problem. Suppose that, in some block $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, a non-terminal X appears only in production p_T and s is a purely-synthesized attribute attached to X (i.e. s depends on no other attribute of X in the TOP).[5] Then, the classical topological sort algorithm may decide to pick $s(X)$ before e is ready for evaluation, in which case the evaluation of $s(X)$ will occur in os'_T. Since $s(X)$ does not occur at all in gos_F, os'_T and os'_F will certainly not be compatible.

Hence, to prove our theorem, we have to impose some further conditions on the dynamic AG and slightly modify the abstract AG that is associated with it. However, these conditions will easily be met or achieved in practice, so this does not reduce the interest of this approach. The next section will present another approach which does not require these additional constraints but requires to modify the ordered sequences generator.

[4]Hence the "black box" name for this approach.

[5]The example in figure 16 below illustrates this case.

The abstract AG that we will hand over to our "black box" is not the one defined above but a slight enrichment of it:

Definition 3.4 (Extended Abstract AG) *The* Extended Abstract Attribute Grammar *for a given Dynamic Attribute Grammar* $DAG = (G = (G_d, G_c, \mathcal{C}oncrete), A, F)$ *is a tuple* $EAAG = (G_a, A_a, F_a)$ *where*

- $G_a = (N_a, T_a, Z_a, P_a)$; $N_a = N_d$;$Z_a = Z_d$; $T_a = T_d$;
- $A_a = \bigcup_{X \in N_a} A_a(X)$, *where* $A_a(Z_a) = A(Z_d)$ *and* $\forall X \in N_a, X \neq Z_a, A_a(X) = A(X) \cup \{\mathtt{x}\}$; $\mathtt{x}$ *is a new boolean inherited attribute;*
- $P_a = \{c.p_d : X_0 \rightarrow X_1 \ldots X_{n_{p_d}} \mid \exists b \in F, ((c, p_d), R) \in \mathcal{R}^b$ *with* $p_d : X_0 \rightarrow X_1 \ldots X_{n_d}\}$
- $F_a = \cup_{p \in P_a} F_a(p)$ *is a set of semantic rules, with* $F_a(p) = R$ *such that* $p = c.p_d$ *and* $\exists b \in F, ((c, p_d), R) \in \mathcal{F}^b_{\mathtt{x}}$, *with* $\mathcal{F}^b_{\mathtt{x}}$ *defined below.*

Definition 3.5 ($\mathcal{F}^b_{\mathtt{x}}$) *For each block* b, $\mathcal{F}^b_{\mathtt{x}}$ *is the set of all semantic rules in* b, *qualified* and modified *by the conjunction (path) of conditions that constrain (enable) them.*

Let $\mathcal{F}_{\mathtt{x}}$ *the transformation which is inductively defined as follows:*

- $\mathcal{F}_{\mathtt{x}}(\langle p, R \rangle) = \{((\varepsilon, p), R)\}$
- $\mathcal{F}_{\mathtt{x}}(\langle R, \langle e, b_T, b_F \rangle \rangle) =$

$$\begin{array}{ll} let & \mathcal{F}_{\mathtt{x}}(b_T) = \bigcup_i ((c_i, p_i), R_i),\ \mathcal{F}_{\mathtt{x}}(b_F) = \bigcup_j ((c_j, p_j), R_j), \\ & V_T = \bigcap_i RHS(p_i),\ V_F = \bigcap_j RHS(p_j),\ V = V_T \cap V_F, \\ & \phi_T = \bigcup_i (((e, true).c_i, p_i), R \cup DP(R_i, e) \cup R_{\mathtt{x}}(p_i, V_T - V, e)), \\ & \phi_F = \bigcup_j (((e, false).c_j, p_j), R \cup DP(R_j, e) \cup R_{\mathtt{x}}(p_j, V_F - V, e)) \\ in & \phi_T \cup \phi_F \end{array}$$

where, given a production p, *a set of names* $V \subset RHS(p)$ *and a condition* e *over* p, $R_{\mathtt{x}}(p, V, e) = \{\mathtt{x}(X) := e \mid X \in V\}$.

Then $\mathcal{F}^b_{\mathtt{x}} = \{((c, p), \mathcal{T}(R)) \mid ((c, p), R) \in \mathcal{F}_{\mathtt{x}}(b)\}$, *where* $\mathcal{T}(R)$ *is derived from* R *by the following transformations:*

- *for each* $X \in RHS(p)$ *such that there is no rule in* R *defining* $\mathtt{x}(X)$, *add the semantic rule* $\mathtt{x}(X) := \mathtt{x}(LHS(p))$ *(when* $LHS(p) = Z_a$ *add* $\mathtt{x}(X) := true$*);*
- *if* $LHS(p) \neq Z_a$, *for each* $s \in S(type_p(LHS(p)))$, *transform its definition* $s(LHS(p)) := exp$ *into the rule* $s(LHS(p)) := dp(exp, \mathtt{x}(LHS(p)))$.

Figure 10 shows the dependence graphs for productions **while** of our example, modified by $\mathcal{F}_{\mathtt{x}}$ (the dependencies related to the $\mathtt{x}$ attribute appear in bold). This example is not very significant, but we can see that, for each non-terminal in the RHS of a production, all its synthesized attributes depend on its $\mathtt{x}$ attribute. This attribute depends either on the value of the condition (if the non-terminal it is attached to is not in the intersection of the productions) or on the value of $\mathtt{x}$ on the LHS of the production.

The differences with the original AAG construction, based on transformation $\mathcal{F}$, are as follows:

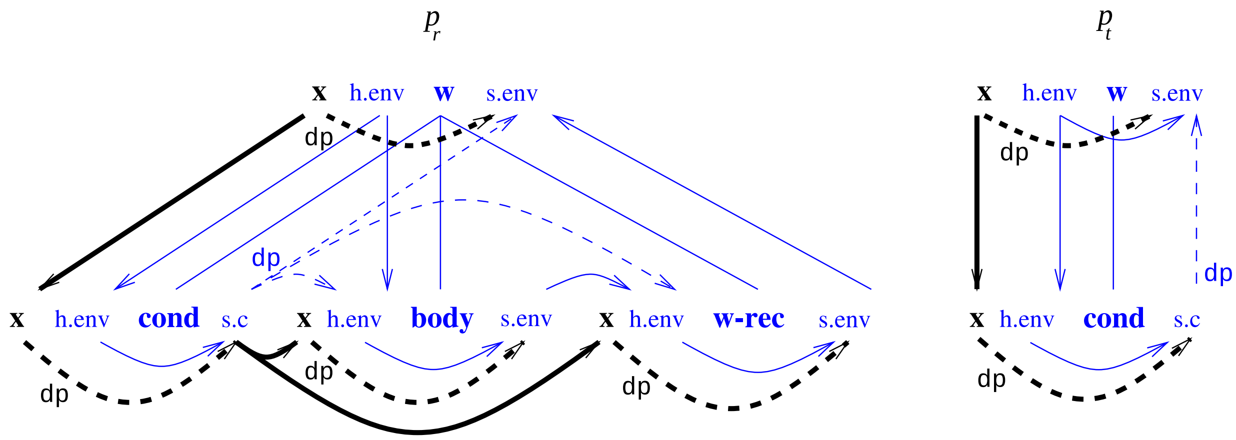

Figure 10: Dependences of the **while** productions with the `x` attribute

1. *Addition of the* `x` *attributes.* $\mathtt{x}(X)$ is an attribute occurrence whose value is irrelevant but which becomes evaluable only when you have the right to reference X; in particular, if X appears in only one of the alternatives of a conditional production, referencing X is only meaningful after the evaluation of the condition, so $\mathtt{x}(X)$ depends on the condition. Thus, although it was introduced mainly for technical reasons (see below), this attribute has the semantic role to concretize the *existence* of the tree node it is attached to.

2. *Addition of dependencies of all synthesized attributes of all non-terminals on their respective* `x` *attributes* (except for the root non-terminal). This ensures that no visit to a node can occur before its `x` attribute is evaluated.

3. *Addition of "control" dependencies.* Consider some non-terminal name X in a simple block $b = \langle R, \langle e, \langle p_T, R_T \rangle, \langle p_F, R_F \rangle \rangle \rangle$. As shown above, if X is not in the intersection of p_T and p_F, we want to forbid that it is visited before the evaluation of the condition e. The rules added by the calls to $R_{\mathtt{x}}$ in transformation $\mathcal{F}_{\mathtt{x}}$ precisely enforce this constraint.

   This, of course, is a restriction, since it will lead to the rejection of some l-ordered DAGs which can be handled by other methods (in particular the *ad hoc* approach of next section). The DAGs which will be rejected are those in which there exists a conditional production and a RHS non-terminal such that the condition depends on some of its attributes (say, those of the first visit) but the definition of some other of its attributes (say, those of the last visit) depends on the condition; in this case, the introduction of the `x` attribute and its dependence on the condition will create a circularity.

   Unfortunately, there is no way to relax this restriction, because the black box approach gives no "source-level" way to distinguish, using only the local dependencies, the attributes of some RHS non-terminal which really depend on the condition from those of the same non-terminal which don't. Hence, we have to enforce that a given RHS name is either entirely in the common part—in which case none of its attributes may depend on the condition—or entirely in one of the specific parts—in which case all of its attributes will depend on the condition.

Note that, when a block contains two nested conditions, the $R_{\mathtt{x}}$ rules generated for the inner one will be correctly processed by the DP transformation for the outer one, so that the corresponding `x` attributes will depend on both conditions.

Now we are ready to prove our theorem.

Proof 3.1 *The demonstration follows the steps of the topological sort algorithm and proceeds by induction on the length of the ordered sequences.*

First of all, notice that $\mathcal{DD}_{p_T}(e) = \mathcal{DD}_{p_F}(e)$. Indeed, if the DAG is well-formed, e is an expression over attribute occurrences which must be well-formed in both p_T and p_F, so the elements of, say, $\mathcal{DD}_{p_T}(e)$, which are exactly the occurrences appearing in the expression e, must also exist in p_F, and vice-versa. So it makes sense to forget about p_T or p_F when reasoning about $\mathcal{DD}(e)$.

Induction hypothesis: Either e is evaluable or $os^n_T = os^n_F \subset W(p_T) \cap W(p_F)$, where n is the length of the ordered sequences.

- *Initial Step: $os^0_T = os^0_F = \emptyset$, QED.*
- *Induction Step: Assume that the induction hypothesis is true after step n.*

  If $\mathcal{DD}(e) \subset os^n_T = os^n_F$, e is evaluable and the induction terminates.

  Otherwise, we note

$$\mathcal{E}_T = \mathcal{E}(os^n_T, p_T) = \{a(X) \in W(p_T) \mid \mathcal{DD}_{p_T}(a(X)) \subset os^n_T, a(X) \notin os^n_T\}$$

  and

$$\mathcal{E}_F = \mathcal{E}(os^n_F, p_F) = \{a(X) \in W(p_F) \mid \mathcal{DD}_{p_F}(a(X)) \subset os^n_F, a(X) \notin os^n_F\}$$

  We want to show that $\mathcal{E}_T = \mathcal{E}_F \subset W(p_T) \cap W(p_F)$, knowing that $os^n_T = os^n_F$ and $\mathcal{DD}(e) \not\subset os^n_T = os^n_F$.

  Let $a(X) \in \mathcal{E}_T$, let's show that $a(X) \in \mathcal{E}_F$ too. There are two cases:

  $a(X) \in W_u(p_T)$**:** *Let's first show that $X \in p_T \cap p_F$.*

  i. If $X = LHS(p_T)$, this is true by definition.

  ii. If $X \in RHS(p_T)$, then $a \in S(type_{p_T}(X))$ and, by virtue of $\mathcal{F}^b_{\mathtt{x}}$, $\mathcal{DD}_{p_T}(a(X)) \neq \emptyset$, since it contains at least $\mathtt{x}(X)$.[6] Since $a(X)$ is ready for evaluation, $\mathtt{x}(X) \in os^n_T = os^n_F$. If $X \notin p_T \cap p_F$, by virtue of the rules in $R_{\mathtt{x}}(p_T, p_T - (p_T \cap p_F), e)$ in the EAAG, $\mathtt{x}(X)$ depends on e, which contradicts the hypothesis that e is not evaluable.

  Since $X \in p_T \cap p_F$ and, by definition, it has the same type in both productions, the T_X used in $\gamma(p_F)$ is the same as in $\gamma(p_T)$. Thus, since, $a(X)$ being an input occurrence, all its dependencies are in T_X, we have $\mathcal{DD}_{p_F}(a(X)) = \mathcal{DD}_{p_T}(a(X)) \subset os^n_T = os^n_F$, which implies that $a(X) \in \mathcal{E}_F$.

  $a(X) \in W_d(p_T)$**:** *Either a is the `x` attribute of some RHS name or the semantic rule $f_{p_T,a,X}$ which defines $a(X)$ is in $R \cup R_T$.*

[6] This is the technical point which mandates the introduction of the `x` attributes.

given $\gamma(p)$ where $p = c.p_d = (e_1, t_1).(e_2, t_2)\dots(e_n, t_n).p_d$ **do**
$os \leftarrow \epsilon$; $i \leftarrow 0$;$S \leftarrow \emptyset$;
repeat
  $i \leftarrow i + 1$;
  if $i = n + 1$ **then** $S \leftarrow W(p)$
  else $S \leftarrow S \cup \mathcal{DD}^+(e_i)$ — *dependency cone of the condition*
  repeat
    compute $\mathcal{E}(os)$;   — *the set of attributes ready for evaluation*
    $a(X_i) \leftarrow \mathcal{P}ick(\mathcal{E}(os) \cap S)$;
    $os \leftarrow os.a(X_i)$;
  until $\mathcal{E}(os) \cap S = \emptyset$
until os is complete.

Figure 11: Conditional topological sort of $W(p)$.

i. If $a = \mathtt{x}$, if $X \notin p_T \cap p_F$, the same reasoning as ii *above leads to a contradiction. So, $X \in p_T \cap p_F$ and $\mathtt{x}(X)$ depends only on $\mathtt{x}(LHS(p_T))$, which is the same as $\mathtt{x}(LHS(p_F))$.*[7]

ii. If $f_{p_T,a,X} \in R_T$, it has been processed by the $DP(R_T, e)$ transformation in $\mathcal{F}_{\mathtt{x}}$, so $\mathcal{DD}(e) \subset \mathcal{DD}_{p_T}(a(X))$; this contradicts our assumption that e is not evaluable, so $f_{p_T,a,X} \in R$, which implies $X \in p_T \cap p_F$ and $f_{p_F,a,X} = f_{p_T,a,X}$. These two facts imply that $a(X)$ depends on the same attributes in p_F as in p_T, because all its dependencies are either in the semantic rule or in (the same) T_X.

In all cases, we conclude that $a(X) \in W_d(p_F)$ with the same definition and dependencies as in p_T, hence $\mathcal{DD}_{p_F}(a(X)) = \mathcal{DD}_{p_T}(a(X)) \subset os^n_T = os^n_F$ and, as above, $a(X) \in \mathcal{E}_F$.

Since the above reasoning is obviously symmetrical in p_T and p_F, we have shown that $\mathcal{E}_T = \mathcal{E}_F \subset W(p_T) \cap W(p_F)$. Then, if $\mathcal{P}ick$ is deterministic, $\mathcal{P}ick(\mathcal{E}_T) = \mathcal{P}ick(\mathcal{E}_F) = a(X) \in W(p_T) \cap W(p_F)$ and $os^{n+1}_T = os^n_T.a(X) = os^n_F.a(X) = os^{n+1}_F \subset W(p_T) \cap W(p_F)$, which concludes the proof of the induction step.

This proof obviously extends to blocks with a decision tree of height greater than one.

3.1.3 The ad-hoc approach

In this section, we present the second approach to ensure that the visit sequences for a simple block are compatible. Unlike the black-box approach, it works with the standard AAG definition but it requires some modification to the topological sort which is used to produce the ordered sequences. More precisely, this modified topological sort is aware of the conditions. The basic idea of this modification is to reduce at each step the set of attribute occurrences to be ordered to only those that are necessary to compute the next condition. The modified algorithm is given in figure 11.

[7] In a more general block, this translates into "$\mathtt{x}(X)$ depends only on conditions higher in the decision tree than e", which apply to both p_T and p_F.

When we use this algorithm on the dependency graphs augmented with the TOPs derived from the original AAG, we have a theorem equivalent to that of the first approach, stating that the visit sequences for a simple block are compatible. To prove this theorem we will use the two following lemmas.

The first one is the most important: it shows that, before the evaluation of some condition, on each production subject to this condition, the topological sort algorithm orders exactly the same set of attribute occurrences.

Lemma 3.2 *For a given block $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, we have $\mathcal{DD}^+_{p_T}(e) = \mathcal{DD}^+_{p_F}(e)$.*

Proof 3.2 *We will reason by recurrence about the transitivity of dependences in $\mathcal{DD}^+$.*

Initial step: $\mathcal{DD}_{p_T}(e) = \mathcal{DD}_{p_F}(e)$

Indeed, if the DAG is well-formed, e is an expression over attribute occurrences which must be well-formed in both p_T and p_F, so the elements of, say, $\mathcal{DD}_{p_T}(e)$, which are exactly the occurrences appearing in the expression e, must also exist in p_F, and vice-versa.

Used occurrence step: $\mathcal{DD}^2_{p_T}(e) = \mathcal{DD}^2_{p_F}(e)$

For each $a(X)$ in $\mathcal{DD}_{p_T}(e)$, $a(X)$ must be an input (or used) occurrence in p_T, and X must appear both in p_T and p_F. Thus, since $type_{p_T}(X) = type_{p_F}(X) = type_b(X)$, the same T_X is used both in $\gamma(p_T)$ and in $\gamma(p_F)$. So we have $\mathcal{DD}_{p_T}(a(X)) = \mathcal{DD}_{p_F}(a(X))$.
Furthermore, since $a(X)$ is an input occurrence in p_T, all its direct dependencies are in T_X: $\forall b(Y) \in \mathcal{DD}_{p_T}(a(X))$, $Y = X$.

Defined occurrence step: $\mathcal{DD}^3_{p_T}(e) = \mathcal{DD}^3_{p_F}(e)$

Now, suppose that, with $a(X)$ as in previous step, there exists an attribute occurrence $b(X)$ in $\mathcal{DD}_{p_T}(a(X))$ such that $\mathcal{DD}_{p_T}(b(X)) \neq \mathcal{DD}_{p_F}(b(X))$.
Since $a(X) \in W_u(p_T)$, $b(X) \in W_d(p_T)$ and the direct dependencies of $b(X)$ (the elements of $\mathcal{DD}_{p_T}(b(X))$) are induced either by the TOP T_X or by the semantic rule defining $b(X)$. The difference between $\mathcal{DD}_{p_T}(b(X))$ and $\mathcal{DD}_{p_F}(b(X))$ cannot come from T_X because the same is used both in $\gamma(p_T)$ and in $\gamma(p_F)$. Hence $b(X)$ must be defined by different semantic rules in p_T and p_F, which implies that $f_{p_T,b,X} \in R_T$.
But then, this semantic rule must have been processed by the $DP(R_T, e)$ transformation in $\mathcal{F}$, so $b(X) \leftarrow e$. However, by hypothesis, we also have $e \leftarrow a(X) \leftarrow b(X)$, so there exists a circularity in $\gamma(p_T)$ and $\gamma(p_F)$. Since the AAG is supposed to be l-ordered, this leads to a contradiction.
Hence, for all $a(X) \in \mathcal{DD}_{p_T}(e)$, for all $b(X) \in \mathcal{DD}_{p_T}(a(X))$, $\mathcal{DD}_{p_T}(b(X)) = \mathcal{DD}_{p_F}(b(X))$.

Recurrence

Since it is equivalent to reason about p_T or p_F, these two last steps can be alternatively applied as long as there exists some unexplored dependencies in $\mathcal{DD}^+_{p_T}(e)$ to show that $\forall n, \mathcal{DD}^n_{p_T}(e) = \mathcal{DD}^n_{p_F}(e)$. Then, since $\mathcal{DD}^+_{p_T}(e)$ is acyclic and finite, we can conclude that $\mathcal{DD}^+_{p_T}(e) = \mathcal{DD}^+_{p_F}(e)$. QED.

The following lemma proves that the algorithm described in figure 11 terminates and that it will never encounter a deadlock situation.

Lemma 3.3 *For all executions of the <u>conditional</u> topological sort, if S^i and os^i are the values of S and os at the end of the i-th inner loop, $1 \leq i \leq n$, we have $S^i = os^i$.*

Proof 3.3

$os^i \subset S^i$ *by construction.*

$S^i \subset os^i$ *by contradiction: Let's assume that, at the end of the i-th inner loop,* $\exists x \in S = \bigcup_{j\leq i} \mathcal{DD}^+(e_j)$ *such that* $x \notin os$.

- *Since the loop has ended,* $\mathcal{E}(os) \cap S = \emptyset$*, which implies that* $x \notin \mathcal{E}(os)$*. By definition of* $\mathcal{E}$ *and the assumption that* $x \notin os$*, we infer that* $\mathcal{DD}(x) \not\subset os$*. Furthermore, by definition of* $\mathcal{DD}^+$ *(notion of transitivity),* $x \in \mathcal{DD}^+(e) \Rightarrow \mathcal{DD}(x) \subset \mathcal{DD}^+(e)$*, which implies* $x \in \bigcup_{j\leq i} \mathcal{DD}^+(e_j) \Rightarrow \mathcal{DD}(x) \subset \bigcup_{j\leq i} \mathcal{DD}^+(e_j)$*. So* $\exists a \in \mathcal{DD}(x), a \notin os, a \in S$.
- *But* $a \notin \mathcal{E}(os)$*, since* $\mathcal{E}(os) \cap S = \emptyset$ *(loop hypothesis). So* $a \in \mathcal{DD}(x)$ *is exactly in the same situation as our hypothetical* x *and we can apply the same reasoning to show that* $\exists b \in \mathcal{DD}^2(x), b \notin os, b \in S$ *and, by induction,* $\forall n > 0, \exists c \in \mathcal{DD}^n(x), c \notin os, c \in S$.
- *But, since* $\gamma(p)$ *is acyclic,* $\exists n_0 > 0, \forall d \in \mathcal{DD}^{n_0}(x), \mathcal{DD}(d) = \emptyset$*, which contradicts the fact that* $d \notin os$*.* QED.

Theorem 3.4 *Given a block* $b = \langle R, \langle e, (p_1, R_T), (p_2, R_F)\rangle\rangle$ *which induces:*

- $p_T = (e, true).p_1$ *and* $p_F = (e, false).p_2 \in P_a$,
- os_T *and* os_F *the ordered sequences generated by the* <u>conditional</u> *topological sort algorithm,*
- $gos_T = \mathcal{G}uard(p_T, os_T) = os'_T.cond_e.os''_T$ *and* $gos_F = \mathcal{G}uard(p_F, os_F) = os'_F.cond_e.os''_F$ *the corresponding guarded ordered sequences,*

if $\mathcal{P}ick$ *is a deterministic function, then* $os'_T = os'_F$.

Proof 3.4 *By lemma 3.2,* $\mathcal{DD}^+_{p_T}(e) = \mathcal{DD}^+_{p_F}(e)$*. Furthermore, since the* <u>conditional</u> *topological sort algorithm runs without deadlock until it exhausts* $\mathcal{DD}^+(e)$ *(lemma 3.3) and* $\mathcal{P}ick$ *is a deterministic function working on identical sets,* $os'_T = os'_F$.

3.1.4 Post-processing

Both methods described above (black-box and ad-hoc) lead the evaluator generator to produce *compatible guarded ordered sequences gos* for all productions $c.p$ of an AAG. In this section we show how to produce a complete evaluator from this collection.

Definition 3.6 (Conditional Ordered Sequence) *For a given* $b \in F$*, let* V^b *be the set of pairs* $(c.p, gos)$ *where* $c.p \in \mathcal{PR}^b$ *and gos is its associated guarded ordered sequence. If all gos in* V^b *are compatible, the* conditional ordered sequence *cos associated with b is the result of the* $\mathcal{M}erge$ *transformation over* V^b*, defined as follows:*

if $V^b = \{(p, os)\}$then $\mathcal{M}erge(V^b) = os$

else *let* V_T *and* V_F *be the two following sets:*

$$V_T = \{(c_i.p_i, gos'_i) \mid (c.p_i, gos_i) \in V^b, c.p_i = (e, true).c_i.p_i, gos_i = os_i.cond_e.gos'_i\}$$
$$V_F = \{(c_j.p_j, gos'_j) \mid (c.p_j, gos_j) \in V^b, c.p_j = (e, false).c_j.p_j, gos_j = os_j.cond_e.gos'_j\}$$

in $\mathcal{M}erge(V^b) = os.\langle e, \langle \mathcal{M}erge(V_T), \mathcal{M}erge(V_F)\rangle\rangle$*, where* $os = os_i \simeq os_j$.

```
h.env(w), h.env(cond), s.c(cond),
⟨ s.c(cond),
    ⟨ h.env(body), s.env(body), h.env(w-rec), s.env(w-rec), s.env(w) ⟩,
    ⟨ s.env(w) ⟩ ⟩
```

Figure 12: Conditional ordered sequence for the **while** block

Since the guarded ordered sequences given in figure 9 are compatible, we can construct the conditional ordered sequence for the **while** block presented in figure 12.

In the same way as, in classical Attribute Grammars, the evaluator generator constructs a *visit sequence* from a given *ordered sequence*, it now constructs a *conditional visit sequence* from a given *conditional ordered sequence*. In order to explain how this conditional visit sequence can be exploited, we will quickly recall the classical notion of evaluators based on the visit sequence paradigm [Kas80, Alb91, Kas91]. There exists one visit sequence per production p, which is a sequence of instructions drawn from the following set:

`begin` i**:** begin the i-th visit to the current node.

`eval` $a(X)$**:** evaluate the attribute occurrence $a(X)$; the attributes on which $a(X)$ depend are guaranteed to be available.[8]

`visit` i, X**:** perform a recursive visit (the i-th one) to son X of the current node ($X \in \mathcal{V}_p$); on that son, fetch the applied production and jump to the `begin` i instruction in the corresponding visit sequence.

`leave` i**:** terminate the current (i-th) visit of the current node and return to its father; continue on the father with the instruction following the `visit` i, X which caused the current visit to begin.

The algorithm producing a (classical) visit sequence from a given ordered sequence is given in figure 13.[9] It makes use of the following auxiliary definition.

Definition 3.7 *Given a production p and a complete ordered sequence os on $W(p)$, let S be some subset of $W(p)$. Then:*

$$Last(os, S) = \begin{cases} \perp & \text{if } S = \emptyset \\ w & \text{otherwise, where } os = os'.w.os'', w \in S, os'' \cap S = \emptyset \end{cases}$$

$$First(os, S) = \begin{cases} \perp & \text{if } S = \emptyset \\ w & \text{otherwise, where } os = os'.w.os'', w \in S, os' \cap S = \emptyset \end{cases}$$

The visit sequences corresponding to the (plain) ordered sequences[10] for the **while** productions are given in figure 14.

[8]This `eval` instruction is often extended to handle *sets* of attribute occurrences which are simultaneously evaluable; this is more convenient when the topological sort algorithm is also able to handle sets—the elements of the TOPs—rather than individual attributes in the graph. In this paper however, all our algorithms deal with individual attributes; so will the `eval` instruction.

[9]The advised reader will have noticed that the last `visit` to some son X may be forgotten is this visit is purely inherited, i.e. if $S_{n_X} = \emptyset$. We believe that such non-productive visits are useless, at least in the case of Dynamic AGs, so we didn't bother to fix this.

[10]The ordered sequences were not given previously but they can be easily derived from their guarded counterparts presented in figure 9.

Given p and os **do**
$n \leftarrow n_{LHS(p)}$ — *number of visits to p*
$vs \leftarrow$ `begin` 1;
repeat
  $a(X) \leftarrow head(os)$;
  if $X = LHS(p) \wedge a \in S_i(X)$ **then**
    $vs \leftarrow vs.$`eval` $a(X)$;
    if $a(X) = Last(os, S_i(X)) \wedge i \neq n$ **then**
      $vs \leftarrow vs.$`leave` $i.$`begin` $i+1$ **fi**
  elsif $X \neq LHS(p) \wedge a \in H_j(X)$ **then**
    $vs \leftarrow vs.$`eval` $a(X)$
  elsif $X \neq LHS(p) \wedge a(X) = First(os, S_j(X))$ **then**
    $vs \leftarrow vs.$`visit` j, X
  fi; $os \leftarrow tail(os)$
until $os = \epsilon$;
$vs \leftarrow vs.$`leave` n.

Figure 13: Algorithm to produce a visit sequence from an ordered sequence

For p_r:
```
begin 1; eval h.env(cond); visit 1, cond;
    eval h.env(body); visit 1, body;
    eval h.env(w-rec); visit 1, w-rec;
    eval s.env(w); leave 1.
```
For p_t:
```
begin 1; eval h.env(cond); visit 1, cond;
    eval s.env(w); leave 1.
```

Figure 14: Plain visit sequences for the **while** productions

```
begin 1; eval h.env(cond); visit 1, cond;
⟨ s.c(cond),
    ⟨   eval h.env(body); visit 1, body;
        eval h.env(w-rec); visit 1, w-rec;
        eval s.env(w); leave 1 ⟩,
    ⟨   eval s.env(w); leave 1 ⟩ ⟩.
```

Figure 15: Conditional visit sequence for the **while** block

The visit-sequence construction algorithm extends in an obvious way to more structured versions of ordered sequences, namely guarded and conditional ordered sequences.

Definition 3.8 (Conditional visit sequence) *For a given block b and its associated conditional ordered sequence cos, the* conditional visit sequence *cvs is the result of the visit-sequence construction algorithm over cos.*

The conditional visit sequence for the **while** block, derived from the conditional ordered sequence in figure 12, is given in figure 15.

The visit sequences are easy to implement as recursive procedures which leave some state information and attribute values at the tree nodes they traverse [Kas91]. Well-known works on storage optimisation [Kas87, JP90] help reduce the total amount of needed memory but also the proportion of attributes which must be stored in the tree. It is important to notice that the most effective of these techniques, which are based on a static analysis of the whole collection of visit sequences [JP90], also apply to our Dynamic AGs (conditional visit sequences) because they don't rely on the sequence selection mechanism.

Another implementation, more appropriate for our future research works, is the *visit function* paradigm [SV91]. An important property of this implementation is that, when we use the above storage optimisation techniques and, as a last resort, the *binding tree* [SV91] technique, *no* attribute needs to be stored in the tree anymore. It is hence quite appropriate for the implementation of Dynamic AGs, in which the physical tree need not be isomorphic to the computation tree or even exist at all.

A last step is required before we can generate visit functions, though. Indeed, in the classical approach, there is one visit function per visit to some non-terminal, which tests the production which is applied at the root of the argument subtree and branches to the appropriate sub-sequence (delimited by `begin` i and `leave` i). In Dynamic AGs, however, the appropriate sub-sequence depends not only on the production applied at the root of the argument subtree (which may not exist at all) but also on the path of conditions which have been evaluated in previous visits. So, when we cut a conditional visit sequence into "slices" corresponding to the various visits to the LHS, we need to reintroduce in each of them the branching code executed in previous visits. This is the purpose of the $\mathcal{L}eave$ transformation defined below, which will produce a *dynamic visit sequence* directly transformable into a collection of visit functions.

Definition 3.9 (Dynamic Visit Sequence) *For a given block and its associated conditional visit sequence cvs, the* dynamic visit sequence *dvs is the result of the $\mathcal{L}eave$ transformation over cvs, defined as follows:*

- if $cond \notin cvs$ then $\mathcal{L}eave(cvs) = cvs$;

- if $cvs = vs.\langle e\langle vs_T, vs_F\rangle\rangle$ *and* $cond \notin vs$
  then $\mathcal{L}eave(cvs) = vs.\mathcal{L}eave\langle e\langle vs_T, vs_F\rangle\rangle$;

- if $cvs = \langle e\langle vs_T, vs_F\rangle\rangle$, *with:*
  $\mathcal{L}eave(vs_T) = vs_T^1.\texttt{leave}.vs_T^2$ *with* $\texttt{leave} \notin vs_T^1$
  $\mathcal{L}eave(vs_F) = vs_F^1.\texttt{leave}.vs_F^2$ *with* $\texttt{leave} \notin vs_F^1$
  then if $vs_T^2 = vs_F^2 = \varepsilon$
  then $\mathcal{L}eave(cvs) = \langle e\langle vs_T^1, vs_F^1\rangle\rangle.\texttt{leave}$
  else $\mathcal{L}eave(cvs) = \langle e\langle vs_T^1, vs_F^1\rangle\rangle.\texttt{leave}.\mathcal{L}eave(\langle e\langle vs_T^2, vs_F^2\rangle\rangle$.

Because of our **while** example is not significant enough to illustrate this transformation (there is only one visit), we show in figure 16 the dependency graphs for two productions of an imaginary abstract AG. These productions depend on a condition over a purely synthesized attribute of a variable common to both productions, `s(W)`. If this condition is true, then p_t is applied, otherwise it is p_f. In figure 17 we present successively the conditional ordered sequence associated with these productions, the corresponding conditional visit sequence (produced by the $\mathcal{V}isit$ transformation) and the dynamic visit sequence (produced by the $\mathcal{L}eave$ transformation).

It is clear how to turn such dynamic visit sequences into collections of visit functions, provided that the values of the various conditions computed in one visit are correctly transmitted to subsequent visits as non-temporary local attributes. Also note that some of these condition values to pass from one visit to the next may be undefined, but this is no problem since all the values which will actually be used in the branching code will have been correctly computed.

This concludes the construction of visit-sequence-based evaluators for Dynamic Attribute Grammars. As for traditional AGs, these evaluators are as efficient as possible. When the dynamic AG is evaluable in one pass, the generated visit functions are the same as what you could write by hand in any language with recursive functions; however, when dependencies are more complicated, hand-writing the evaluator is close to impossible, unless you use some sort of delayed evaluation mechanism—e.g. lazy evaluation of functional programs—, but then our eager evaluators are more efficient. See [PDRJ95] for a longer discussion of this topic.

4 Conclusion

In this paper we have argued that in the term "Attribute Grammar" the notion of *grammar* does not necessarily imply the existence of an underlying tree, and that the notion of *attribute* does not necessarily mean decoration of a tree. We have presented Dynamic Attribute Grammars, a new, simple extension to the Attribute Grammar formalism which allows the full exploitation of the power of this observation. They are consistent with the general ideas underlying Attribute Grammars, hence we retain the benefits of the results and techniques that are already available in that domain.

Our goal in providing these extensions to the Attribute Grammar formalism is to bring this powerful tool into a larger context of usefulness and applicability. The Attribute Grammar programming style (declarative and structured) and existing Attribute Grammar techniques (static analysis) become more general under this extended view and reveal themselves as complementary to other formalisms such as functional programming or inference rule programming [PDRJ95].

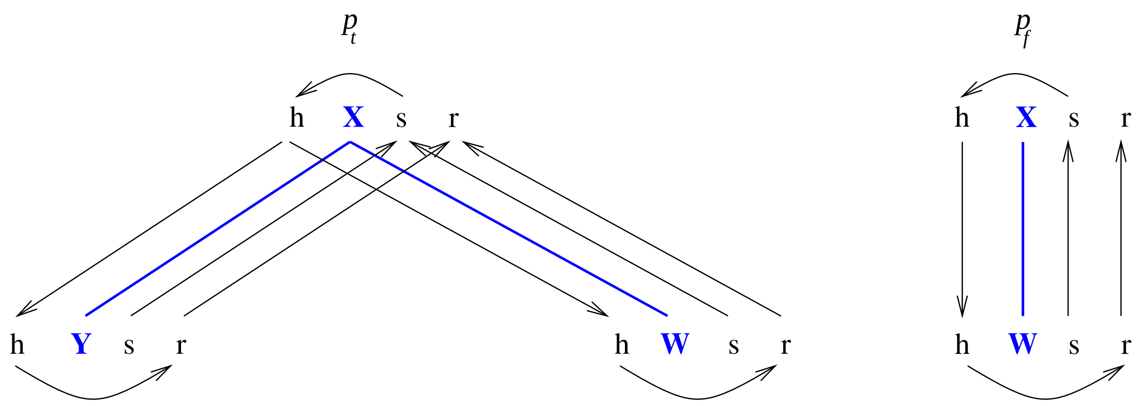

Figure 16: An example of dependence graph

s(W),
⟨ *cond*, — *condition over* s(W)
⟨ s(Y), s(X), h(X), h(Y), h(W), r(Y), r(W), r(X) ⟩,
⟨ s(X), h(X), h(W), r(W), r(X) ⟩ ⟩

(a) The conditional ordered sequence

```
begin 1; visit 1, W;
⟨ cond,
    ⟨   visit 1, Y; eval s(X); leave 1;
        begin 2; eval h(Y); eval h(W); visit 2, Y;
            visit 2, W; eval r(X); leave 2; ⟩,
    ⟨   eval s(X); leave 1;
        begin 2; eval h(W); visit 2, W; eval r(X); leave 2; ⟩ ⟩
```

(b) The conditional visit sequence

```
begin 1; visit 1, W;
⟨ cond ,
    ⟨ visit 1, Y; eval s(X); ⟩,
    ⟨ eval s(X); ⟩ ⟩
leave 1;
begin 2;
⟨ cond,
    ⟨ eval h(Y); eval h(W); visit 2, Y; visit 2, W; eval r(X); ⟩,
    ⟨ eval h(W); visit 2, W; eval r(X); ⟩ ⟩
leave 2;
```

(c) The dynamic visit sequence

Figure 17: Example of $\mathcal{V}isit$ and $\mathcal{L}eave$ transformations

This approach is of practical interest because, as we have shown in detail, the mechanisms necessary to support Dynamic Attribute Grammars were already part of the FNC-2 system, which has proved its usefulness on real applications; this made their implementation easy. It is also promising because it opens the way to the application of good results developed for Attribute Grammars to other programming paradigms.

References

[Alb91] Henk Alblas. Attribute evaluation methods. In Alblas and Melichar [AM91], pages 48–113.

[AM91] Henk Alblas and Bořivoj Melichar, editors. *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.*, Prague, June 1991. Springer-Verlag.

[Att89] Isabelle Attali. *Compilation de programmes TYPOL par attributs sémantiques.* PhD thesis, Université de Nice, April 1989.

[Boy96] John Boyland. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, January 1996.

[CFZ82] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes (i and ii). *Theor. Comp. Sci.*, 17(2 and 3):163–191 and 235–257, 1982.

[DJ90] Pierre Deransart and Martin Jourdan, editors. *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes in Comp. Sci.*, Paris, September 1990. Springer-Verlag.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lect. Notes in Comp. Sci.* Springer-Verlag, August 1988.

[Eng84] Joost Engelfriet. Attribute grammars: Attribute evaluation methods. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[Far83] Rodney Farrow. Covers of attribute grammars and sub-protocol attribute evaluators. Technical report, Department of Comp. Sc., Columbia University, New York, NY, September 1983.

[Far86] Rodney Farrow. Automatic Generation of Fixed-point-finding Evaluators for Circular, but Well-defined, Attribute Grammars. In *ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 85–98, Palo Alto, CA, June 1986.

[Fil83] Gilberto Filé. Interpretation and reduction of attribute grammars. *Acta Informatica*, 19:115–150, 1983. See also: memorandum 359, Onderafdeling der Informatica, Tech. Hogeschool Twente (1981).

[Jou92] Martin Jourdan. *Des bienfaits de l'analyse statique sur la mise en œuvre des grammaires attribuées.* Mémoire d'habilitation, Département de Mathématiques et d'Informatique, Université d'Orléans, April 1992.

[JP90] Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Deransart and Jourdan [DJ90], pages 29–45.

[JP93] Martin Jourdan and Didier Parigot. *The* FNC-2 *System User's Guide and Reference Manual.* INRIA, Rocquencourt, 1.9 edition, 1993.

[JPJ+90] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *ACM SIGPLAN '90 Conf. on Programming Languages Design and Implementation*, pages 209–222. White Plains, NY, June 1990. Published as ACM SIGPLAN Notices, volume 25, number 6.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980. See also: Bericht 7/78, Institut für Informatik II, University Karlsruhe (1978).

[Kas87] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24(6):633–652, November 1987.

[Kas91] Uwe Kastens. Implementation of visit-oriented attribute evaluators. In Alblas and Melichar [AM91], pages 114–139.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2(2):127–145, June 1968.

[KW76] Ken Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *3rd ACM Symp. on Principles of Progr. Languages*, pages 32–49. Atlanta, Ge, January 1976.

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

[Par88] Didier Parigot. *Transformations, évaluation incrémentale et optimisations des grammaires attribuées: le système FNC-2.* PhD thesis, Université de Paris-Sud, Orsay, May 1988.

[PDRJ95] Didier Parigot, Étienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. Rapport de recherche 2662, INRIA, October 1995.

[PDRJ96] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *Journées Francophones des Langages Applicatifs 96*, pages 263–279, Val-Morin, Québec, January 1996. Aussi dans les *Actes des journées du GDR Programmation 95.*

[SV91] S. Doaitse Swierstra and Harald H. Vogt. Higher Order Attribute Grammars. In Alblas and Melichar [AM91], pages 256–296.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399