This paper appears in *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, ed. by H. Kuchen and S. D. Swierstra, Aachen (FRG), September 24–27, 1996, pp. 122–136, vol. 1140 of *Lect. Notes in Comp. Sci.*, Springer-Verlag.

Dynamic Attribute Grammars
(*Extended Abstract*)

Didier PARIGOT, Gilles ROUSSEL, Martin JOURDAN and Étienne DURIS*

INRIA and Université de Marne-la-Vallée

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*, 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).

Keywords: Attribute Grammars, static analysis, implementation, dynamic semantics, applicative programming.

1 Introduction and Related Work

Attribute Grammars were introduced thirty years ago by Knuth [15] and, since then, they have been widely studied [7, 6, 2, 17]. An Attribute Grammar is a declarative specification that describes how attributes (variables) are computed for rules in a particular grammar (i.e., it is syntax-directed). They were originally introduced as a formalism for describing compilation applications and were intended to describe how to decorate a 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

* Gilles Roussel is with Université de Marne-la-Vallée, 2, allée du Promontoire, 93166 Noisy-le-Grand, France; e-mail: roussel@univ-mlv.fr. The other authors are with INRIA, Projet OSCAR, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France; e-mail: {Didier.Parigot, Martin.Jourdan, Etienne.Duris}@inria.fr; Web page: http://www-rocq.inria.fr/oscar/FNC-2/.

– 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.

In spite of that, Attribute Grammar specifications are still not as widely used as they could be. We believe that one of the main reasons for this is their lack of expressiveness, which is due to the fact that, because of their historical roots in compiler construction, the notion of (physical) tree was considered as the only way to direct computations. Some works have attempted to respond to this problem by proposing extensions to the classical Attribute Grammar formalism, for instance Circular Attribute Grammars [9], Multi-Attributed Grammars [3], Higher-Order Attribute Grammars [21] or Conditional Attribute Grammars [4]. Our own work [18, 19] has similarities with the latter two (like with HOAGs, the computation tree is not isomorphic to the input tree, and like with CAGs, attribute values can influence the choice of semantic rules to compute) but our approach differs in important respects. First, for us, the notion of *grammar* does *not* necessarily imply the existence of a (physical) *tree* and, in fact, our evaluators can work *without any tree*. Secondly, our implementation technique is a simple derivation of the traditional visit-sequence-based evaluation paradigm and does not require the construction of any additional piece of tree.

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) [5], whereas the semantic rules describe the computations associated with this scheme (data flow).

It is very important to observe that all the theoretical and practical results on Attribute Grammars 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 particular, this applies to the algorithms for constructing efficient evaluators for various subclasses of Attribute Grammars and the global static analysis methods [10, 2, 17].

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 (possibly empty) 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*, decision trees for productions and their semantic rules.

These extensions result in 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 [12, 11].

An informal, example-based comparison of Dynamic Attribute Grammars with other programming paradigms appears in [18], 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 would be too long to present here anyhow. This paper is a much shortened version of [20], in which the interested reader will find more details, more formalism and all the proofs.

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 "behavior" as a given DAG (the *Abstract Attribute Grammar*, or AAG, associated with the DAG). The second section demonstrates how to use classical AG-implementation techniques to produce efficient, visit-sequence-based evaluators for DAGs.

2 Dynamic Attribute Grammars

2.1 Recalls on Classical Attribute Grammars

Definition 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, $p : X_0 \to X_1 \dots X_n$ 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 (Attribute Grammar). An *Attribute Grammar* is a tuple $AG = (G, A, F)$ where:

- $G = (N, T, Z, P)$ is a context-free grammar;
- $A = \bigcup_{X \in N} H(X) \uplus S(X)$ is a set of attributes, with $H(X)$ the *inherited* attributes of $X \in N$ and $S(X)$ the *synthesized* ones;
- $F = \bigcup_{p \in P} F(p)$ 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 $p : X_0 \to X_1 \dots X_n$ 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 whereas, 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.

Definition 3 (Production). Let $\mathcal{V}$ be a universal finite set of *names*. A production $p : X_0 \to X_1 \ldots X_n$ 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. $type_p : \mathcal{V}_p^{\oplus} \to N \cup T$, where $\mathcal{V}_p^{\oplus} = \{X_0\} \cup \mathcal{V}_p$, is a function which associates to each name a unique type in the set of non-terminals and terminals, such that $type_p(X_0) \in N$.

In the sequel of this paper, we will use the clearest of our two notations for a production—$p : X_0 \to X_1 \ldots X_n$ 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 $RHS(p) = \mathcal{V}_p$.
- $W_u(p)$, the set of *input* or *used* attribute occurrences in p, and $W_d(p)$, the set of *output* or *defined* attribute occurrences, are defined as usual; $W(p) = W_u(p) \cup W_d(p)$.

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.

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.

Definition 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 $G_c = (N_c, T_c, Z_c, P_c)$ and a function $\mathcal{C}oncrete : P_d \times \mathcal{V} \to (P_c \times \mathcal{V}) \cup \{\bot\}$, where:

1. $N_c \subseteq N_d$; $T_d = T_c$; if G_c is not empty[2] then $Z_d = Z_c$.
2. $\forall p_d \in P_d$, we have:
   i. $\forall X \in \mathcal{V}_{p_d}^{\oplus}$, $type_{p_d}(X) \in (N_d - N_c) \Rightarrow \mathcal{C}oncrete(p_d, X) = \bot$;
   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 $type_{p_d}(LHS(p_d)) = type_{p_c}(LHS(p_c))$;
      - $\forall X \in RHS(p_d)$, $type_{p_d}(X) \in N_c \Rightarrow \exists Y \in \mathcal{V}_{p_c}^{\oplus}$ such that $\mathcal{C}oncrete(p_d, X) = (p_c, Y)$ and $type_{p_d}(X) = type_{p_c}(Y)$.

[2] In completely tree-less applications, such as the factorial function [18], G_c is empty and $\mathcal{C}oncrete$ maps any element to $\bot$.

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.

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).

A dynamic production p_d is either purely abstract or associated with 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. 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 [18].

Condition 3 stems from the constraint that, for two productions with the same LHS type and the same associated $\mathcal{C}oncrete$[3], 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.

Our running example in this paper will be to define (an excerpt of) the *dynamic semantics* of a programming language with a DAG describing an interpreter. This application is out of the reach of traditional AGs and is the basis for our translation of denotational semantics into DAGs [16]. Fig. 1 presents the structure of the **while** statement as part of a grammar couple $(G_d, G_c, \mathcal{C}oncrete)$. `STAT,COND` $\in N_d \cup N_c$ respectively represent statements and boolean conditions. `name:TYPE` means that `TYPE` is the type of `name` and `name_d=name_c` means that $\mathcal{C}oncrete(p_d, \texttt{name\_d}) = (p_c, \texttt{name\_c})$.[4] $p \in P_c$ 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 represent the recursive and termination behaviours of a **while** structure.

A semantic rules block is a conditional structure (decision tree) which defines all the dynamic productions that are applicable at a same point (either associated with the same concrete production or the same purely dynamic non-terminal, see the constraints in definition 7 below), their semantic rules and the conditions specifying how to choose between them.

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

[4] where p_d and p_c are unambiguously defined by the context.

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 loop=while:STAT`

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

Fig. 1. Part of a grammar couple for the **while** statement

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

Fig. 2. The semantic rules block for the **while** statement

Definition 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.

Fig. 2 presents the semantic rules block describing the denotational-like semantics of the **while** statement. Attributes names are prefixed by `h.` for inherited, and `s.` for synthesized. The attribute `env` represents the execution environment (store, etc.) of a statement and `s.c` carries the value of the 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 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:

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

The $\mathcal{R}^b$ set for our **while** example can be derived from Fig. 3 below by forgetting about the DP transformation introduced before definition 9.

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

- $G = (G_d, G_c, \mathcal{C}oncrete)$ is a grammar couple;
- $A = \bigcup_{X \in N_d} H(X) \uplus S(X)$ is a set of attributes;
- 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, as defined above;
  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:
     - $X \in (N_d - N_c) \Rightarrow b_i = b_j$;
     - $X \in N_c \Rightarrow (b_i = b_j \Leftrightarrow \mathcal{C}oncrete(p) = \mathcal{C}oncrete(q))$.

A Dynamic Attribute Grammar describes a function taking as arguments:

- values for all the inherited attributes of the start symbol (since these are not banned), 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, based on the notion of consistently attributed dynamic (virtual) trees, is the topic of our present work; 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 [18] will help the reader intuitively grasp the semantics and operation of a DAG.

2.3 Abstract Attribute Grammars

We claimed earlier that Dynamic Attribute Grammars could be implemented using the same techniques as classical AGs. The basic idea is simple [10]:

1. build from the given DAG a classical AG which has the same "behavior" (syntax—i.e., recursion scheme—and dependencies—i.e., data flow);
2. generate the evaluator for this classical AG;
3. transform this evaluator so that it correctly implements the original DAG.

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

Let $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$ be the simplest form of a (conditional) block. Basically, the productions and semantic rules in the AAG which will reproduce the behavior of this block are, on one hand, p_T associated with the rules in $R \cup R_T$ and, on the other hand, $(p_F, R \cup R_F)$. 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 8 (Abstract AG). 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.

In this definition, $c.p_d$ is just a name for a production in AAG which encodes its origins in DAG: the production p_d and sequence of guards c. For instance, for our **while** example, the two productions in the AAG in Fig. 3 below are the same as p_r and p_t in Fig. 1, up to the production names.

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 9 ($\mathcal{F}^b$ set). 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:

- $\mathcal{F}^{\langle p,R\rangle} = \{((\varepsilon, p), R)\}$
- $\mathcal{F}^{\langle R,\langle e,b_{true},b_{false}\rangle\rangle} =$
  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 $\cup_i(((e, true).c_i, p_i), R \cup DP(R_i, e))$
  $\bigcup \cup_j(((e, false).c_j, p_j), R \cup DP(R_j, e))$.

This is nearly the same as the flattened form $\mathcal{R}_b$, except that it makes explicit the "control dependencies" on the conditions. Fig. 3 presents the productions and modified semantic rules in the AAG for the **while** statement.

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

Fig. 3. Productions and modified semantic rules in the Abstract AG for the **while** statement

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 observation is the basis for the formal definition of the semantics of a DAG, on which we are working.

3 Visit-Sequence-Based Implementation

In this section, we show how to produce evaluators for dynamic AGs based on the visit sequence paradigm [14, 8, 1]. This 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 [12] (for the reason just mentioned and for their versatility); and they are the easiest to transform into functions or procedures, which gives a basis for our studies on the relationships between AGs and functional programming [18].

The presentation in this version of the paper is very informal, relying almost entirely on pictures and examples. The formal definitions, the theorems and their proofs appear in [20].

Fig. 4 illustrates the generation process and introduces the various objects it manipulates. It proceeds as follows (the figure numbers refer to the corresponding objects for the **while** example):

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) $\{T_X \mid X \in N\}$ of the attributes of each non-terminal. Since not all AGs are l-ordered, this may fail for some dynamic AGs. For the **while** example we have $T_{\texttt{STAT}} = \{\texttt{h.env}\}\{\texttt{s.env}\}$ and $T_{\texttt{COND}} = \{\texttt{h.env}\}\{\texttt{s.env}, \texttt{s.c}\}$.

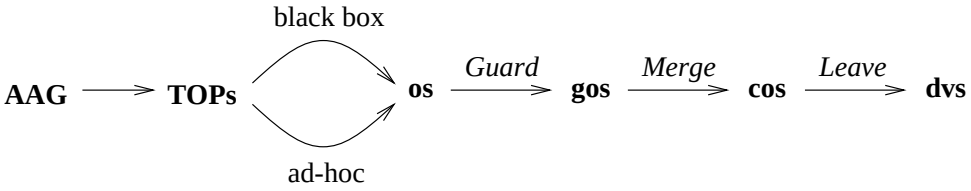

Fig. 4. The basic idea

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

Fig. 5. Guarded ordered sequences for the **while** productions

2. Using these TOPs, we generate, for each of the productions of the AAG, a separate visit sequence represented by an *ordered sequence os*, i.e. an ordered subset of $W(p)$ such that the total order on *os* respects the partial order $\gamma(p)$ of the augmented dependence graph $D(p)[T_{X_0}, T_{X_1}, \ldots, T_{X_n}]$. *os* is derived from $\gamma(p)$ by topological sort and it is easy to construct the visit sequence from *os*. Note that this step and the previous one are exactly the same as for a traditional AG.
3. Each production in the AAG corresponds to some "guarded production" *c.p* in the dynamic AG. We hence reintroduce in each ordered sequence marks for the evaluation and test of the various conditions (guards) of the dynamic production it corresponds to; for each condition, in the order defined by the path in the decision tree, this occurs as soon as all the attribute occurrences on which it depends are available. This leads to *guarded ordered sequences* (Fig. 5).
4. We then merge all the guarded ordered sequences corresponding to the same block, so as to obtain a single *conditional ordered sequence* structured just like the decision tree of the block (Fig. 6). To make this possible, we have to make sure that these visit sequences are "compatible", i.e. that, for a simple

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

Fig. 6. Conditional ordered sequence for the **while** block

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

Fig. 7. Conditional visit sequence for the **while** block

block of the form $\langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$, the parts 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. This point is discussed further below.

5. We transform each conditional ordered sequence into a *conditional visit sequence* by the same process as the traditional transformation of an ordered sequence into a visit sequence (Fig. 7).
6. We cut each conditional visit sequence in "slices" corresponding to the various visits to the LHS node, so as to make each slice a separate *visit function*, and we reintroduce at the beginning of each such function the branching code executed in previous visits.

Because our **while** example is not significant enough to illustrate the last step (there is only one visit), we show in Fig. 8 the (augmented) dependency graphs for two productions of an imaginary abstract AG. These productions depend on a condition over a purely synthesized attribute of a name common to both productions, `s(W)`. If this condition is true, then p_t is applied, otherwise it is p_f. In Fig. 9 we present successively the conditional ordered sequence associated with these productions, the corresponding conditional visit sequence and the dynamic visit sequence.

Let us get back to the notion of *compatibility*, briefly touched upon in step 4 above. Consider a simple block $b = \langle R, \langle e, (p_T, R_T), (p_F, R_F)\rangle\rangle$ 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: evaluate the attributes "before" the condition; evaluate the condition; 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. Hence the whole construction relies on the following theorem:

Theorem 10. *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,*

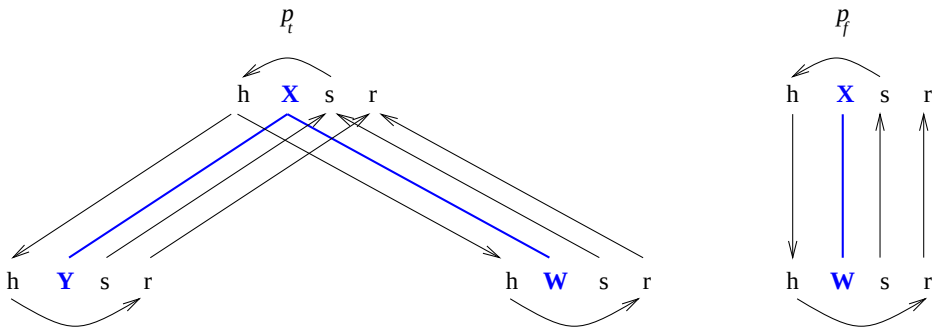

Fig. 8. An example of dependence graph

```
    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

Fig. 9. Example of $\mathcal{V}isit$ and $\mathcal{L}eave$ transformations

given $\gamma(p)$ where $p = c.p_d = (e_1, t_1).(e_2, t_2) \ldots (e_n, t_n).p_d$ **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.

Fig. 10. Conditional topological sort of $W(p)$

– $gos_T = \mathcal{G}uard(p_T, os_T) = os'_T.cond_e.os''_T$ *and* $gos_F = \mathcal{G}uard(p_F, os_F) = os'_F.cond_e.os''_F$ *the corresponding guarded ordered sequences,*

if the choice function used in the topological sort is deterministic, then $os'_T = os'_F$.

In [20], we present two approaches to the construction of ordered sequences and the proof of this compatibility theorem: the first one (*black box*) uses the classical construction of ordered sequences, without any modification, but requires that we start with a slightly more rigid AAG[5] than the one presented earlier; the second one (*ad hoc*) starts with the standard AAG but requires that the construction of ordered sequences (topological sort) is aware of the conditions (see Fig. 10: attributes to evaluate are picked in the "dependency cone" of the successive conditions).

The final form of our evaluators is based on the *visit function* paradigm [21]. An important property of this implementation is that, when we use classical, static storage optimization techniques [13] and, as a last resort, the *binding tree* technique [21], *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.

The last step before the generation of these visit functions, namely the construction of dynamic visit sequences (Fig. 9), is required to account for the fact that the visit-sequence selection mechanism of Dynamic AGs is richer than that of classical AGs: the latter only depends on the production which is applied at the root of the visited subtree, whereas the former (possibly) uses this information but also the conditions. So, when we cut a conditional visit sequence into "slices" corresponding to the various visits to the LHS, we need to reintroduce

[5] The added constraint enforces that no son which is not entirely in the "intersection" of p_T and p_F is visited before the evaluation of condition e. This may lead to the rejection of a few l-ordered, meaningful DAGs.

in each of them the branching code executed in previous visits. This assumes of course that the values of the various conditions computed in one visit are correctly transmitted to subsequent visits as non-temporary local attributes.

This concludes the construction of visit-sequence-based evaluators for Dynamic AGs. Like 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 one 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 one uses some sort of delayed evaluation mechanism—e.g. lazy evaluation of functional programs—, but then our eager evaluators are more efficient. See [18] 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 AG 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. Its declarative and structured programming style and existing static analysis techniques become more general under this extended view and reveal themselves as complementary to other formalisms such as functional programming or inference rule programming [18].

This approach is of practical interest because, as we have shown, 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

1. Henk Alblas. Attribute evaluation methods. In Alblas and Melichar [2], pages 48–113.
2. Henk Alblas and Bořivoj Melichar, editors. *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.*, Prague, June 1991. Springer-Verlag.
3. Isabelle Attali. *Compilation de programmes TYPOL par attributs sémantiques.* PhD thesis, Université de Nice, April 1989.

4. John Boyland. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, January 1996.
5. 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.
6. 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.
7. 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.
8. Joost Engelfriet. Attribute grammars: Attribute evaluation methods. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.
9. 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.
10. 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.
11. Martin Jourdan and Didier Parigot. *The* FNC-2 *System User's Guide and Reference Manual.* INRIA, Rocquencourt, 1.9 edition, 1993.
12. 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.
13. Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Deransart and Jourdan [6], pages 29–45.
14. Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980. See also: Bericht 7/78, Institut für Informatik II, University Karlsruhe (1978).
15. Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2(2):127–145, June 1968.
16. Stéphane Leibovitsch. Relations entre la sémantique dénotationnelle et les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1996.
17. Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
18. Didier Parigot, Étienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. Rapport de recherche 2662, INRIA, October 1995. `ftp://ftp.inria.fr/INRIA/publications/RR/RR-2662.ps.gz`.
19. 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.*
20. Didier Parigot, Gilles Roussel, Martin Jourdan, and Étienne Duris. Dynamic attribute grammars. Rapport de recherche 2881, INRIA, May 1996. `ftp://ftp.inria.fr/INRIA/publications/RR/RR-2881.ps.gz`.
21. S. Doaitse Swierstra and Harald H. Vogt. Higher Order Attribute Grammars. In Alblas and Melichar [2], pages 256–296.