INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Attribute Grammars: a Declarative Functional Language

Didier PARIGOT , Gilles ROUSSEL , Etienne DURIS , Martin JOURDAN

N° 2662

Octobre 1995

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

Attribute Grammars:
a Declarative Functional Language

Didier Parigot , Gilles Roussel , Etienne Duris , Martin Jourdan

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet charme

Rapport de recherche n° 2662 — Octobre 1995 — 13 pages

Abstract: Although Attribute Grammars were introduced thirty years ago, their lack of expressiveness has resulted in limited use outside the domain of static language processing. In this paper we show that it is possible to extend this expressiveness. We claim that Attribute Grammars can be used to describe computations on structures that are not just trees, but also on abstractions allowing for infinite structures. To gain this expressiveness, we introduce two new notions: *scheme productions* and *conditional productions*. The result is a language that is comparable in power to most first-order functional languages, with a distinctive declarative character.

Our extensions deal with a different part of the Attribute Grammar formalism than what is used in most works on Attribute Grammars, including global analysis and evaluator generation. Hence, most existing results are directly applicable to our extended Attribute Grammars, including efficient implementation (in our case, using the Fnc-2 system `http://www-rocq.inria.fr/charme/FNC-2/`).

The major contribution of this approach is to restore and re-emphasize the intrinsic power of Attribute Grammars. Furthermore, our extensions call for new studies on applying to functional programming the analysis and implementation techniques developed for Attribute Grammars.

Key-words: Attribute grammars, functional programming, static analysis, global analysis

(Résumé : tsvp)

Ce travail a été partiellement financé par le projet ESPRIT #5399 “Compare”.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Les Grammaires Attribuées : un langage fonctionnel déclaratif

Résumé : Bien que les Attribute Grammars aient été introduites il y a trente ans, leur manque de pouvoir d'expression les a confinées dans le domaine du traitement des langages de programmation. Dans cet article, nous montrons qu'il est possible d'étendre cette expressivité. Nous soutenons que les Attribute Grammars peuvent être utilisées pour décrire des calculs sur des structures qui ne sont pas uniquement des arbres, mais aussi des formes abstraites permettant de décrire des structures infinies. Afin d'atteindre cette expressivité, nous avons introduit deux nouvelles notions : les *schémas de productions* et les *productions conditionelles*. 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.

Nos extensions ne remettent pas en cause les bases du formalisme des Attribute Grammars sur lesquelles reposent la plupart des travaux concernant celles-ci, en particulier l'analyse statique et la génération d'évaluateurs. Ainsi, les résultats existants peuvent être appliqués directement à nos Attribute Grammars étendues, entre autre ceux permettant une implantation efficace (dans notre cas, en utilisant notre système FNC-2 `http://www-rocq.inria.fr/charme/FNC-2/`).

L'intérêt de ces extensions est de redonner aux Attribute Grammars leur expressivité intrinsèque. De plus, elles nous permettent d'envisager de nouveaux axes de recherche en comparant nos techniques d'analyses à celles qui ont été développées dans des formalismes de même expressivité.

Mots-clé : Grammaires Attribuées, Programmation Fonctionnelle, Analyse Statique, Analyse Globale

1 Introduction

Thirty years ago Attribute Grammars were introduced by Knuth [Knu68] and since then they have been widely studied [DJL88, DJ90, AM91]. 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 driven). Attribute Grammars 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 structures 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 [Jou92].

In spite of that, Attribute Grammar specifications are still not as widely used as they could be. Because of the historical roots of Attribute Grammars 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. First, there are few application domains, apart from compilation, that naturally provide a tree as the main input. Furthermore, in the absence of such a tree, the computation model is quite weak: you cannot even define the factorial function with a pure, classical Attribute Grammar. This lack of expressiveness and disagreement on basic issues inside the Attribute Grammars community have combined to cast a shadow over the good results that have been accumulated over thirty years regarding Attribute Grammars. Many were led to think that Attribute Grammars are so simple and limited that they were unusable for the development of real applications and methods for their static analysis could not be transposed to other formalisms.

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] or Multi-Attributed Grammars [Att89]. 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 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 [CFZ82] (control flow), 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 propose two linguistic extensions which comply with this view:

- *conditional productions* that allow attribute values to influence the flow of control, and
- *scheme productions* that allow to describe recursion schemes independently from any physical structure and/or exhibiting a different combination of elements of a physical structure.

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, JPG93].

The availability of this "new" programming language is, however, not the main achievement of our work. Indeed, the actractiveness of a particular language w.r.t. its competitors is a very subjective matter. In our opinion, the justification of our new language is that it is the concrete form for exploiting the static analysis techniques developed for Attribute Grammars, which are of high interest in their own sake. By freeing these results from the narrow frame of traditional Attribute Grammars, we believe that they will find applications in other domains.

The remainder of this article is divided into three sections. In the first of these, the new notions of conditional productions and scheme productions are introduced using some small examples, intentionally taken from outside the compilation area. The next section demonstrates why these extensions do not require modifications to the techniques used for evaluator generation. The last section enumerates the benefits of using extended Attribute Grammar specifications. Most of the presentation is intentionally informal.

2 An External View of Attribute Grammars

In this section we describe classical Attribute Grammars and our proposed extensions using short, informal and incomplete examples. Each of these is presented using an Attribute Grammar specification in Olga, a functional specification (the CAML dialect of ML) and inference rules (Typol [Des88]).[1] The ML and Typol examples are not meant to be a basis for *comparison* but rather an aid for intuitively *understanding* the semantics of our extensions; lack of space prevents us from introducing the formalisms in detail here.

2.1 The Classical View of Attribute Grammars

We now introduce the classical formalism of Attribute Grammars using a simple example called *bird* [Bir84, Joh87]. The purpose is to take as input a binary tree with integer leaves, and to give as output the same binary tree where all leaves are replaced by the minimum value of the input tree leaves.

Before the specification of the Attribute Grammar, we specify the tree type, or abstract syntax. This abstract syntax is also translated into an ML type.[2]

Abstract Syntax:
```
AXIOM = axiom;
axiom -> TREE;
TREE = fork tip;
fork -> TREE TREE;
tip -> int;
```

ML Type:
```
type TREE = fork of TREE * TREE | tip of int;;
```

The Attribute Grammar specification of the example follows. The attribute `$min` is used to compute the minimum value of the input tree leaves, `$n` propagates the minimum value throughout the tree, and `$tree` computes the new output tree.

[1] In this paper, programming by inference rules encompasses logic programming.
[2] The consistency of the Attribute Grammar specification requires the addition of a root node type that is not necessary in ML.

Attribute Grammar specification:

```
attribute grammar bird (TREE) : (TREE) is
attribute
    synthesized $min (TREE) : int;
    synthesized $tree (TREE) : TREE;
    synthesized $tree_axiom (AXIOM) : AXIOM;
    inherited $n (TREE) : int;

where axiom -> T use
    $tree_axiom := axiom($tree(T)); $n (T) := $min (T);
end where;

where tip -> M use
    $min := M; $tree := tip ($n(tip)) ;
end where;

where fork -> L R use
    $min := if ($min(L) <= $min(R))
            then $min(L) else $min(R) end if;
    $tree := fork ($tree(L), $tree(R));
    $n(L) := $n(fork); $n(R) := $n(fork);
end where;
end grammar;
```

A simple syntactic transformation [Joh87] results in the following single ML function:

ML:

```
let bird t = t1
    where (t1, n) = repmin t n
    where rec repmin t n = match t with
       tip (m) -> (tip (n), m) |
       fork (L, R) -> let (t1, m1) = repmin L n
                      and (t2, m2) = repmin R n
                      in
                        (fork (t1, t2),
                         (if m1 < m2 then m1 else m2));;
```

Finally, here is the same example specified with inference rules:

Typol:

```
program bird is
use tree;
import min(integer, integer -> integer) from aux;

judgement |- AXIOM : TREE;
judgement integer |- TREE : integer, TREE;

 n |- T : n, t
----------------
|- axiom(T) : t;

n |- tip M : M, tip n;

n |- L : m1, t1 & n |- R : m2, t2 & min(m1, m2 -> m)
----------------------------------------------------
       n |- fork(L, R) : m, fork(t1, t2);
end bird;
```

Note that the ML definition requires lazy evaluation and that the Typol definition requires unification; in both cases, this is because there is a "circularity" at the root of the tree on n, the minimum value at the leaves, which is both a result and an argument.

Our FNC-2 system can produce evaluators in several different languages, including ML [Bât95]. The ML form of the evaluator that is automatically generated from the Attribute Grammar specification given above is given below. This form does not require lazy evaluation, but inferring it requires more intellectual effort. In the Attribute Grammar specification, no order of evaluation was specified, and this is the essence of its declarativeness and attractiveness: it is *not necessary* to write it in a specific way to make it efficient. Indeed, the evaluator generator has automatically determined that the minimum should first be evaluated, and then the tree should be constructed. These two passes break the "circularity". In this simple example the evaluation order is easy to find by hand, but this is not always the case.

ML program derived from Attribute Grammar:

```
let rec replace t m = match t with
    tip (n) -> tip(m) |
    fork(L, R) -> fork ((replace L m), (replace R m));;

let rec tmin t = match t with
    tip (n) -> n |
    fork (L, R) -> let m1 = (tmin L), m2 = (tmin R) in
                (if m1 < m2 then m1 else m2);;

let bird t = replace t (tmin t);;
```

2.2 Conditional Productions

In this section we introduce the notion of *conditional production*. The syntax proposed here, reminiscent of *guards*, is very simple; we will later discuss its restrictions and possible extensions. These new productions allow the specification of different sets of semantic rules on a single production. The set of rules to be applied to a specific instance of a production is chosen dynamically. Essentially, we add predicates over attribute values to supplement the patterns used to select which attribute computations to apply.

In this next example, there are two different sets of semantic rules on the `fork` operator. The value of the `$min` attribute governs the choice between the two sets. This specification has the same semantics as the first *bird* specification for the computation of the `$min` attribute. We require that there always be a default rule, indicated by the label `[]`, which is applied if none of the other conditional productions can be applied.

Conditional Production:

```
where fork[$min(L) <= $min(R)] -> L R use
    $min := $min(L);
end where;

where fork[] -> L R use
    $min := $min(R);
end where;
```

Other examples will appear later in the paper. The main use of conditional productions is in combination with scheme productions, but note here that they have the interest to decouple the dependence schemes associated with the various alternatives and possibly make them simpler.

2.3 Scheme Productions

In the classical theory of Attribute Grammars, the notion of grammar implies the existence of some tree. We now extend this notion with a new notion of *scheme productions*. These scheme productions describe in a general way how to dynamically connect sets of semantic rules; in the classical Attribute Grammar formalism this connection was driven exclusively by the input tree.

Together with the notion of conditional productions, the scheme productions allow the expression of solutions to many different kinds of recursive problems that could not be handled in the classical formalism. The presentation of this extension uses the well known problem of the Towers of Hanoi.

Attribute Grammar Specification:
```
attribute grammar HANOI ($n:int; $from, $to, $work:tower) :
    ($r:list-of-moves) is
where ht [$n=1]-> use
    $r(ht) := move($from(ht),$to(ht));
end where;

where hr [] -> H1 H2 use
    $n(H1) := $n(hr) - 1;       $n(H2) := $n(hr) - 1;
    $from(H1) := $from(hr);     $from(H2) := $work(hr);
    $to(H1) := $work(hr);       $to(H2) := $to(hr);
    $work(H1) := $to(hr);       $work(H2) := $from(hr);
    $r(hr) := append($r(H1), move($from(hr),$to(hr)), $r(H2));
end where;
```

Figure 1: Towers of Hanoi specified with an Attribute Grammar

ML:
```
let rec hanoi n from to work =
    if n=1 then
      (move from to)
    else
      append (hanoi (n-1) from work to)
             (move from to)
             (hanoi (n-1) work to from);;
```

In addition to the classical productions (with the implicit presence of a tree), we now allow scheme productions to be associated to some left-hand-side non-terminal which can be viewed as a type. Here, a `HANOI` type is defined, with scheme productions that **only** describe recursive calls; this purely virtual scheme does **not** define any physical tree.

Scheme production example:
```
HANOI = with scheme
    hr -> HANOI HANOI
    ht ->
end scheme;
```

This scheme shows two patterns. In the first pattern, the recursive case, `HANOI` is defined in terms of two different uses of itself. In the second pattern, the termination case of the recursion, `HANOI` makes use of no other definitions. This scheme represents only uses of type definitions. As we will see when the example is worked out, both the recursive and the terminating clauses of this scheme will make use of classical attribute value calculations.

The Olga specification for this example is given in Fig. 1. A condition on the number of disks chooses between the recursive and the termination cases. Note that, since `HANOI` is a pure scheme type, it has no (tree) value and hence cannot (and will not) be used to drive the evaluator flow; the explicit condition will be the sole discriminant.

This example can be expressed in Typol, as shown in Fig. 2.

2.4 Using Extensions With Tree Productions

With classical Attribute Grammars it is not possible to accurately describe the dynamic semantics of a language containing loops (such as a **while** statement). In this example, we extend a classical production with a scheme that allows the introduction of circularities into the tree. In this scheme, an element in the RHS of the production (the second `Expr`) is a reference to (the LHS of) the production itself. In the second scheme, we have “forgotten” about the loop body, which allows us to avoid specifying and evaluating its attributes.

Typol:
```
program hanoi is
use HANOI;
judgement integer, integer, integer, integer |- HANOI : list;

1,from,_,to |- _ : move(from,to);

n-1,from,work,to |- H1 : l' & n-1,work,to,from |- H2 : l''
------------------------------------------------------------
n,from,to,work |- hr(H1,H2) : l' . move(from, to) . l'';

end hanoi;
```

Figure 2: Towers of Hanoi specified in Typol

Abstract Syntax:
```
Expr = while;
while -> Cond Expr
with scheme
    wr -> Cond Expr Expr=wr
    wt -> Cond
end scheme;
```

We now describe part of an interpreter for the `while` instruction as an Attribute Grammar specification. The attributes `$s.env` and `$h.env` contain the execution environment; `$c` contains the value of the condition.

Incomplete Attribute Grammar Specification:
```
where wt[$c (C) = false] -> C use
    $h.env(C) := $h.env(wt);
    $s.env(wt):= $h.env(wt);
end where;
where wr[] -> C E W use
    $h.env(C) := $h.env(wr);
    $h.env(E):= $h.env(wr);
    $h.env(W):= $s.env(E);
    $s.env(wr):= $s.env(W);
end where;
```

We can express this same example in ML and in Typol:

ML:
```
let rec interp_expr x env = match x with
  while (C E) ->
    if (interp_cond C env) then
      let env1 = interp_expr E env in
        interp_expr x env1
    else
      env;;
```

Typol:
```

     env |- C: false
------------------------
env |- while (C,E) : env;

env |- C: true & env |- E: env1 & env1 |- while (C,E) : env2
------------------------------------------------------------
               env |- while (C,E) : env2;
```

As a last example, we show that we can describe the *double* functional, which was deemed impossible in [DJL88, p.46]:

Attribute Grammar:
```
STMT = double ...;
double -> p:STMT
  with scheme dd -> p p;
```

with a simple flow of information across the two occurrences of `p`.

3 An Internal View of Attribute Grammars

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.

The basic idea is to feed the evaluator generator with a version of the input Attribute Grammar in which the extensions have been replaced by their effect on the control flow. In the case of conditional productions, this is achieved by removing the conditions and uniquely renaming each alternative. In the case of scheme productions, no transformation is needed at all.

After the evaluator generator has ran on this extensions-less Attribute Grammar, it is necessary to reintroduce in the generated evaluator the semantics of the extensions.

For scheme productions, this simply means to replace references to nodes in the RHS of a scheme by references to their corresponding "real" nodes (if any). In the case of circular schemes, such as the `while` example, this means that a given node may be visited twice or more, or even indefinitely (which is exactly the purpose of this extension). This mandates that the evaluator be able to store all attributes outside the tree. FNC-2, in particular, uses sophisticated analysis techniques based on attribute lifetimes that allow most attributes to be stored in global variables or stacks [JP90b], resorting to a more systematic technique such as binding trees [SV91] for the very few that remain.

Conditional productions introduce a new way to choose between the various alternatives applicable at a given point. To take this into account, the back-end should generate a new production selector. The selection of a production which has been renamed from a conditional production occurs when the pattern match implied by the underlying production succeeds, and the condition specified in the extended form is satisfied. Of course, this is only possible when the condition is evaluable before—i.e. does not depend on—any attribute computed in the production.

This restriction is clearly not acceptable. First, it strongly reduces the expressive power.[3] Second, it is at odds with the declarative character of the specification because it forces the writer to think about dependencies and evaluation order.

To solve these problems we propose another syntactic form of conditional productions, called a *factored form*. A possible factored form of the `while` example is given in Fig. 3. There are two important features to notice. First, the conditions do not control all semantic rules, which allows the conditions to depend on attributes computed in these independent rules (or on input attributes that depend on them). Thus, condition evaluation may occur anywhere in the attribute evaluation process and not only at production-selection time. Second, the precedence of the various conditions is now explicit.

From this factored form it is possible to produce a flattened form close to the type we have been using, with a simple distributive transformation. The back-end can then collect the evaluation code generated for all alternatives and can rearrange it into a common part followed by a classical conditional statement; this allows the evaluation of the condition to occur elsewhere than at the beginning.

The validity of this process has been demonstrated by its implementation into FNC-2, which took only a few man-weeks and indeed left the evaluator generator strictly untouched.

[3]For instance, the example of the `while` loop does not meet it.

Factored form:
```
where while -> C E use
  $h.env(C) := $h.env(while)
  if [$c(C) = false] then scheme wt -> C use
    $s.env(wt):= $h.env(wt)
  end scheme
  else scheme wr -> C E W use
    $h.env(E):= $h.env(wr);
    $h.env(W):= $s.env(E);
    $s.env(wr):= $s.env(W);
  end scheme;
end where;
```

Figure 3: Factored form of the `while` example in Olga

4 Advantages of This Approach

4.1 A Declarative Programming Language

The extensions to the classical Attribute Grammar formalism proposed in this paper improve its expressiveness and result in a language that is comparable in power to a first-order-only version of ML or Typol. While it does not subsume the latter, this "new" language is attractive because it retains all the pleasant properties of programming with Attribute Grammars:

- Extended specifications are still declarative, so that programmers do not need to worry about evaluation order.
- They are well structured because the classical structuring paradigm based on productions is extended to scheme productions and conditional productions.

These features are, in our opinion, quite useful. In particular, they allow a writer to specify an application in an incremental way: from a kernel specification the complete application can grow and be tested little by little, without damaging readability [JP90a].

The examples given in this paper are, unfortunately for us, too short to show the effectiveness of Olga for developping large programs. For instance, most copy rules of the form `$a(X):= $a(Y)` can be automatically generated and omitted from the program; this allows the developer to concentrate on important computations rather than on simple information transfer [JP90a].

This programming language is already available in our FNC-2 system [JPJ+90, JPG93]. FNC-2 provides powerful mechanisms for composition and reusability [BJPR93, Bel93, Rou94]. It is a very flexible system which can generate exhaustive or incremental attribute evaluators in C, Lisp, and ML, running on monoprocessor or shared-memory multiprocessor machines [Mar94] and using sophisticated memory optimization techniques. With companion processors, FNC-2 can generate complete applications. But it is also an open system that can interface with many other tools. This allows a developer to fully exploit the complementarity of Attribute Grammars with other programming paradigms (in particular, functional programming), by letting him select the most appropriate language and system for each part of his application.

At first sight, allowing attributes to carry functional values (closures) and applying them in semantic rules seems totally independent from the basic Attribute Grammar formalism and the work described here. This is a first step towards making Olga a fully functional language, but not the topic of this paper.

Given that the choice of a programming language over another is a very subjective matter, it is not our aim here to convert every ML or Typol programmer to Olga. In fact, it is clear to us that the Attribute Grammar formalism is restricted, in that it is just a syntactic abstraction of some function or inference rule, and it does not subsume these other formalisms and tools.

Besides, in our opinion, the real interest of our work is not the language itself but the computation model that underlies it. This model is that of classical Attribute Grammars, and our work was "just" to extend its applicability.

4.2 Cross-fertilization

The "new power" of the Attribute Grammar formalism enabled by our extensions leads us to re-examine the results of thirty years of research and practice, and in particular the static analysis methods, in a larger context. The basic observation is that, when Attribute Grammars are considered as a functional dialect, the evaluator generation process can be seen as a functional program transformation system; this was exemplified in section 2, with the lazy-ML version of *bird* and the ML program generated from the Attribute Grammar version. In this respect, the simplicity of the Attribute Grammar formalism, which has often been derided in the past as a weakness, now appears as a strength. Indeed, it allows many static analysis techniques to come up with very precise results (and even with exact results in the case of the plain circularity problem [Jou92]).

For instance, the evaluator generator in FNC-2 performs an extensive analysis of all relations (dependencies) among all attributes to determine the most efficient way to compute them. This kind of analysis can be compared to optimization techniques such as strictness analysis in lazy functional languages. In addition, memory optimization techniques based on attribute lifetime analysis [JP90b] can be seen as an interprocedural transformation that allows to reduce the memory requirements of a functional program. In both cases, these techniques have been extensively studied in the context of Attribute Grammars and have been validated through the implementation and use of systems such as FNC-2 and LIGA.

We have recently begun to explore other tracks. For instance, preliminary studies [Dur94] suggest that descriptional composition [GG84, RJP94, Rou94] and deforestation [Wad88] lead to equivalent results, although they operate quite differently. Furthermore, attribute lifetime analysis gives new insights on the update-in-place problem for functional programs [Dur94, DPJ94]. More work is needed to decide whether this approach is actually fruitful.

Other classical Attribute Grammar paradigms that deserve an interpretation in terms of functional programs include incremental evaluation [Rep84, Par88], parallel evaluation [Jou91, Mar94] and our work on reusability and genericity [BJPR93, Bel93, Rou94].

It is clear, however, that we are only at the beginning of this road, and there are presently many more questions than answers in this area. We are strongly interested in actively studying this topic in the future.

5 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 two simple extensions to the Attribute Grammar formalism that allow 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.

This approach is of practical interest because the mechanisms necessary to support the extensions proposed here 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 developped for Attribute Grammars to other programming paradigms.

Acknowledgments We owe a lot to Isabelle Attali for long and hard discussions which resulted in the ideas presented here, to Uwe Assmann, Pierre Weis and Christian Queinnec for their encouragements and to David Barnard for improving the clarity and linguistic quality of a previous version of this paper.

Final words By the way, we can now write the following:

Factorial function defined by an Attribute Grammar:
```
FACT = with scheme
  fac-t ->
  fac-r -> FACT
end scheme;

attribute grammar FACT($n:int) : ($r:int) is
where fac-t [$n <= 1] -> use
  $r := 1;
end where;
where fac-r [] -> F use
  $n(F) := $n - 1;
  $r := $n * $r(F);
end where;
```

References

[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.

[Bât95] Gilles Le Bâtard. Réalisation dans le système FNC-2 d'un traducteur vers ML. Rapport de stage de maîtrise, IFI, Université de Marne-la-Vallée, July 1995.

[Bel93] Carole Le Bellec. *La généricité et les grammaires attribuées.* PhD thesis, Département de Mathématiques et d'Informatique, Université d'Orléans, November 1993.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversal of data. *Acta Informatica*, 21:239–250, 1984.

[BJPR93] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lect. Notes in Comp. Sci.*, pages 123–136, Tallinn, 1993. Springer-Verlag.

[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.

[Des88] Thierry Despeyroux. Typol: a Formalism to Implement Natural Semantics. Research report 94, INRIA, 1988.

[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.

[DPJ94] Étienne Duris, Didier Parigot, and Martin Jourdan. Mises à jour destructives dans les grammaires attribuées. Draft, September 1994.

[Dur94] Étienne Duris. Transformation de grammaires attribuées pour des mises à jour destructives. Rapport de DEA, Université d'Orléans, September 1994.

[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.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Conf. on Functional Prog. Languages and Computer Architecture*, volume 274 of *Lect. Notes in Comp. Sci.*, pages 154–173, Portland, OR, September 1987.

[Jou91] Martin Jourdan. A Survey of Parallel Attribute Evaluation Methods. In Alblas and Melichar [AM91], pages 234–255.

[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.

[JP90a] Martin Jourdan and Didier Parigot. Application Development with the FNC-2 Attribute Grammar System. In Dieter Hammer and Michael Albinus, editors, *Compiler Compilers '90*, volume 477 of *Lect. Notes in Comp. Sci.*, pages 11–25. Springer-Verlag, Schwerin, October 1990.

[JP90b] Catherine Julié and Didier Parigot. Space Optimization in the FNC-2 Attribute Grammar System. In Deransart and Jourdan [DJ90], pages 29–45.

[JPG93] Martin Jourdan, Didier Parigot, and Tamás Gaál. *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.

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

[Mar94] Bruno Marmol. *La parallélisation et l'optimisation mémoire dans l'évaluation des grammaires attribuées.* PhD thesis, Université d'Orléans, November 1994.

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

[Rep84] Thomas Reps. *Generating Language-Based Environments.* MIT Press, 1984.

[RJP94] Gilles Roussel, Martin Jourdan, and Didier Parigot. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *Lect. Notes in Comp. Sci.*, pages 52–67, Edinburgh, April 1994.

[Rou94] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

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

[Wad88] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, March 1988.

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