Attribute Grammars and Folds : Generic Control Operators

Étienne Duris, Didier Parigot, Gilles Roussel, Martin Jourdan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Attribute Grammars and Folds : Generic Control Operators

Etienne DURIS, Didier PARIGOT, Gilles ROUSSEL, Martin JOURDAN

N° 2957

Août 1996

THÈME 2

ISSN 0249-6399

INRIA
ROCQUENCOURT

Attribute Grammars and Folds : Generic Control Operators

Etienne DURIS, Didier PARIGOT, Gilles ROUSSEL, Martin JOURDAN

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oscar

Rapport de recherche n ° 2957 — Août 1996 — 26 pages

Abstract: Generic control operators, such as *fold*, have been introduced in functional programming to increase the power and applicability of data-structure-based transformations. This is achieved by making the structure of the data more explicit in program specifications.

We argue that this very important property is one of the original concepts of attribute grammars. In this paper, we informally show the similarities between the fold formalism and attribute grammar specifications. We also compare their respective method to eliminate the intermediate data structures introduced by function composition (notion of deforestation or fusion): the normalization algorithm for programs expressed with folds and the descriptional composition of attribute grammars.

Rather than identify the best way to achieve deforestation, the main goal of this paper is merely to intuitively present two programming paradigms to each other's supporting community and provide an unbiased account of their similarities and differences, in the hope that this leads to fruitful cross-fertilization.

Key-words: Attribute grammars, static analysis, functional programming, structure-directed programming, program transformation, deforestation.

(Résumé : tsvp)

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

Grammaires Attribuées et Folds : Opérateurs de Contrôle Génériques

Résumé : Les opérateurs de contrôle génériques tels que *fold* ont été introduits dans la programmation fonctionnelle pour augmenter la puissance et le champs d'application des transformations basées sur la structure des données. Ceci est obtenu en rendant la structure des données plus explicite dans la spécification des programmes.

Nous considérons que cette caractéristique fondamentale est l'un des concepts de base des grammaires attribuées. Dans cet article, nous exposons informellement les similitudes entre le formalisme du fold et la spécification des grammaires attribuées. Nous comparons également leurs méthodes respectives d'élimination des structures intermédiaires introduites par la composition de fonctions (notion de déforestation ou de fusion) : l'algorithme de normalisation pour les programmes exprimés à l'aide de folds et la composition descriptionnelle pour les grammaires attribuées.

Plutôt que de déterminer la meilleure façon d'effectuer la déforestation, le but principal de cet article est simplement de présenter intuitivement chacun de ces deux paradigmes de programmation à la communauté qui soutient l'autre, et de décrire objectivement leurs similitudes et leur différences, dans l'espoir que cela conduise à des fertilisations croisées fructueuses.

Mots-clé : Grammaires attribuées, analyse statique, programmation fonctionnelle, programmation dirigée par la structure, transformation de programmes, déforestation.

1 Introduction

For the last few years, the functional-programming community has been studying various techniques to optimize programs, in particular to eliminate "useless" intermediate data structures occurring in function composition. The first approaches were based on partial evaluation techniques [Ses, CD93]. More recently, a symbolic approach, called *deforestation*, has been proposed [Wad88]; it is mainly based on the "fold and unfold" transformations first introduced in [BD77]. Since this first article, several different formalisms have been introduced to extend the power of these functional-program transformations [MFP91, FSS92, GLJ93, SF93, FSZ94]. Of particular interest to us is the *fold* generic control operator, as presented by Sheard and Fegaras [SF93], and its *normalization algorithm* based on the *fold promotion theorem*, although they have been recently subsumed by more abstract formalisms and more powerful transformations, such as the generic theorem for algebraic types [TM95] and monads [Feg96]. Interestingly enough, many of these approaches are based on the *structure-directed programming style*, which is attracting a growing interest in the functional-programming community.

On the other hand, the problem of eliminating intermediate data structures occurring in function composition has been studied rather extensively in the context of another well-known structure- or syntax-directed programming paradigm, namely Attribute Grammars (AGs) [Knu68, DJL88, AM91]; this lead to the *descriptional composition*[1] algorithm for AGs [GG84, GGV86, Gie88], which, given two AGs such that the result of the first is the argument to the second, produces a single AG which has the same semantics as the composition of the original AGs but without actually constructing the intermediate structure.

The generic control operators such as *fold* allow to capture patterns of recursion for large classes of types in a uniform way, so they look very much like attribute grammars specifications. Furthermore, it seems that deforestation is very similar to descriptional composition; at least, they have the same goal.

The aim of this paper is thus to draw a comparison between the *fold* control operator and AGs, as structure-directed programming paradigms, regarding both their expressive power and their support for the automatic elimination of intermediate data structures. In this first, intuitive approach to this kind of comparison, we will deal mostly with program schemes based on (concrete) types; hence, we prefer to stay with the formalism of [SF93]—instead of using a more powerful theory—and the traditional AG formalism. Future work will deal with more abstract structure-directed programming formalisms [FSZ94, TM95, Feg96] and the recently introduced Dynamic Attribute Grammars [PDRJ96, PDRJ95, PRJD96a, PRJD96b], which are liberated from the existence of a physical structure.

At this point, it is necessary to present Attribute Grammars a little more precisely. They were introduced thirty years ago by Knuth [Knu68] and, since then, they have been widely studied [DJL88, DJ90, AM91, Paa95, JP]. An AG 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, AGs were recognized as having these two important qualities:

- they have a natural *structural decomposition* that corresponds to the syntactic structure of the language, and

[1] This phrase was coined by the inventors of the technique, Ganzinger and Giegerich, so we will use it in this paper, but it is not very aesthetically pleasing; in other works, we have used the phrase "meta-composition".

- 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 AG specification?" Most of the research in the area has hence focused on the automatic production of efficient code without losing expressiveness. This has resulted in the identification of classes of AGs of various size and inherent efficiency, and of characterization algorithms for each of these classes [DJL88, Eng84, Alb91]. For reasons of efficiency, these works on evaluation methods have mostly avoided to use higher-order functions.[2] In this context, the problem of eliminating intermediate data structures introduced by function composition has been studied by Ganzinger and Giegerich and solved by their descriptional composition algorithm [GG84, GGV86, Gie88, Rou94, RJP94, RPJ95].

Furthermore, we have extended the notion of attribute grammars into Dynamic Attribute Grammars [PDRJ96, PDRJ95, PRJD96a, PRJD96b]. Our view of the grammar underlying an AG 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). This allows to use all static analysis and implementation techniques developed for traditional AGs in the much larger context of Recursive Program Schemes [CFZ82, vdM94].

In this article, we first show that grammar definitions underlying AGs can be compared to the algebraic type definitions used in fold functions. The first-order fold formalism allows to specify a large class of programs over an algebraic type definition. We will show that, from a specification point of view, these programs can easily be translated into a purely-synthesized attribute grammar over the corresponding underlying grammar. Since the semantics (computation of the result) of a fold function is given by the definition of a *functor* [SF93], the latter can be considered as a particular evaluator for synthesized attribute grammars (with only one attribute). In the end, we will show that, in this particular context of first-order fold functions and purely-synthesized AGs, the normalization algorithm and descriptional computation are nearly equivalent (they produce the same results).

Next, we will consider the most general case, with higher-order folds on one hand and AGs with inherited attributes on the other hand; both provide for more powerful deforestation. Indeed, there exist some first-order fold programs which are not directly normalizable. However, the $\mathcal{F}_{\mathcal{G}}$ transformation of [SF93] (similar to the continuation-passing-style transformation CPS) allows to transform a not normalizable first-order fold program into a second-order one, which is normalizable with a second-order promotion theorem [SF93].

We will see that using higher-order attributes and semantic rules is not a natural approach with attribute grammars; rather, inherited attributes allow to specify a top-down computation over a recursive structure. On the other hand, the fold formalism cannot express the notion of inherited attributes, and must represent top-down computations as the yet-to-be-computed result of a recursive function which is one of the parameters (by introducing higher-order functions).

Of course, descriptional composition applies to nearly[3] any AG (incl. with inherited attributes) without introducing more complex operators or requiring auxiliary transformations.

[2] There exists a classical transformation which transforms any AG into a purely-synthesized AG with higher-order attributes and semantic rules [Knu68, CM79] but, to our knowledge, it has never been used in practice.

[3] The few restrictions are quite natural and required for the result to be well-defined.

Descriptional composition uses the explicit structure information in AGs to symbolically perform the composition, by means of semantic rules projections [GG84], and eliminate the intermediate data structure. With the same goal, the normalization algorithm takes advantage of the fold definition (the functor) to produce a more efficient transformation. The differences in the results obtained with both methods are not essential nor a real penalty; rather, they might lead to interesting cross-fertilization. The main motivation of this article is then to show that, in these two completely different domains, the same motivations have lead to similar results. Moreover, it provides some possible ways to cross-fertilize both approaches for solving open problems.

Most of the presentation is intentionally informal because the techniques and formalisms used in these two domains (functional programming and attribute grammars) are really different. It seems to us that presenting this comparison in a formal way would seriously hamper the intuitive understanding of the similarities. This is why we don't give the formal definition of fold, but we base our presentation and comparison on the Sheard and Fegaras paper [SF93], and several examples presented in the sequel are copied *verbatim* from [SF93].

The remainder of this article is divided in two sections. The first one successively presents and compares algebraic type definitions with context-free grammars (CFG)[4], the fold notation with the attribute grammar notation, the first-order fold definition based on the notion of functor with attribute evaluation for purely-synthesized attribute grammars, and finally the Normalization Algorithm (NA) for fold functions with the Descriptional Composition (DC) of attribute grammars. The second section presents two approaches allowing more powerful program transformations; first presented intuitively, the second-order fold approach and the inherited attribute approach are then described with their essential differences and their respective deforestation methods; finally, we discuss possible extensions of this work.

2 First-order Folds and S^1 Attribute Grammars

This section informally presents the notion of first-order fold and the notion of attribute grammar. First, we show that an algebraic type definition is very close to a context-free grammar (see also [CM79, VM82, Far92]). Second, in the first-order context, we show that first-order folds and purely-synthesized attribute grammars are *equivalent* generic control operators. Afterwards, we compare the two deforestation techniques these formalisms support (the Normalization Algorithm and Descriptional Composition).

2.1 Algebraic Types and Context-free Grammars

Unlike algebraic type definitions, the AG formalism has no polymorphic (type) feature; this is due to their historical context (compilation, language-based environments, ...), in which the objects at the leaves of a structure (parse tree) are always strings. AGs are based on the *Context-Free Grammar* notion, and in a CFG definition, the terminals which represent the types of the leaves are fixed, i.e. they cannot be parameters of the CFG. But, in fact, when a CFG is considered as describing general trees, without any reference to a text to parse, the terminals play exactly the same role of type variables as in an algebraic type, in the sense that they don't take part in the recursive definition of the type. In the sequel, we assume that this

[4]See also the Algebraic Definition of AGs [CM79].

restriction of the CFG notion is relaxed. The appropriate extensions of the CFG definition induce no real problems for the AG theory.

The family of *fold* generic control operators are defined in [SF93] over mutually recursive types defined as follows:

Definition 2.1 (Mutually Recursive Types)

$$\bigwedge_{s\in[1\ldots r]} T_s(\alpha_1,\ldots,\alpha_p) = \begin{array}{l} C_1^{T_s}(t_{1,1},\ldots,t_{1,m_1}) \\ \mid \quad \ldots \\ \mid \quad C_n^{T_s}(t_{n,1},\ldots,t_{n,m_n}) \end{array}$$

is a set of mutually recursive types*, where* $T_1,\ldots,T_r$ *are the types being defined,* $\alpha_1,\ldots,\alpha_p$ *denote type variables,* C_i *are the names of constructor functions, and* $t_{i,j}$ *are either type variables (in* $\alpha_1,\ldots,\alpha_p$*) or instantiations of sums-of-products types.*

For such types, the equivalent *pseudo-CFG* can be constructed as follows:

Definition 2.2 (Pseudo-Context-Free Grammar) *Given a set of mutually recursive types as in definition 2.1, the* pseudo-Context-Free Grammar $G = (N, T, P)$ *defines the same type, where:*

- $N = \{T_s \mid s \in [1\ldots r]\}$ *is the set of* non-terminal *symbols;*
- $T = \{\alpha_i \mid i \in [1\ldots p]\}$ *is the set of* terminal *symbols;*
- $P = \{C_i^{T_s} \rightarrow t_{i,1},\ldots,t_{i,m_i}\}$ *is the set of* productions.

Note that this definition of is obviously not complete to define a real CFG: it lacks the specification of a start symbol $Z \in N$. Our explanation for this lack of start symbol in a pseudo-CFG and a description of its replacement by the *root argument* and *profile* notions will be given in section 2.2.1 below.

In a production $C_i^{T_s} \rightarrow t_{i,1},\ldots,t_{i,m_i}$, which really is a constructor function, all symbols $t_{i,j}$ are type variables viewed as terminal or non-terminal occurrences. In addition, $C_i^{T_s}$ is a type variable just as any other $t_{i,j}$ and, in the sequel, it will sometimes be noted $t_{i,0}$. Still, by abuse of notation, $C_i^{T_s}$ will be also used to denote the name of this production in P (we will use a more accurate notation when necessary).

Fig. 1 presents some examples of algebraic types; the productions of their corresponding pseudo-CFG definitions appear in Fig. 2.

2.2 First-order Folds and Synthesized Attribute Grammars

2.2.1 Folds *vs.* AGs as notations

In [SF93], some classical functions are defined over the simple recursive type *list* using the generic control operator *fold* for this type; we reproduce the definition of *length* in Fig. 3 and the one of *append* in Fig. 5.

The two lambda-expressions in Fig. 3 are called *accumulating functions* and represent the computations to be performed over each constructor of the type *list*: one for the *Nil* constructor, with no parameter, and the other for the *Cons* constructor, with two parameters. These parameters are provided by the generic definition of fold^{list}, which will be presented in Def. 2.4.

$$
\begin{array}{lll}
\text{list}(\alpha) & = & \text{Nil} \mid \text{Cons}(\alpha,\text{list}(\alpha)) \\
\text{tree}(\alpha) & = & \text{Tip}(\alpha) \mid \text{Node}(\text{tree}(\alpha),\text{tree}(\alpha)) \\
\text{int} & = & \text{Zero} \mid \text{Succ}(\text{int}) \\
 & & \\
\text{exp}(\alpha,\beta) & = & \text{Var}(\alpha) \\
 & & \mid \text{Let}(\text{dec}(\alpha,\beta),\text{exp}(\alpha,\beta)) \\
 & & \mid \text{Apply}(\text{exp}(\alpha,\beta),\text{exp}(\alpha,\beta)) \\
\bigwedge \quad \text{dec}(\alpha,\beta) & = & \text{Val}(\beta,\text{exp}(\alpha,\beta)) \\
 & & \mid \text{Fun}(\mathit{string},\beta,\text{exp}(\alpha,\beta))
\end{array}
$$

Figure 1: Examples of algebraic type definitions

For type list*:*
Cons $\rightarrow \alpha$ list
Nil $\rightarrow$
For type tree*:*
Node $\rightarrow$ tree tree
Tip $\rightarrow \alpha$
For type int*:*
Succ $\rightarrow$ int
Zero $\rightarrow$

For types exp *and* dec*:*
Let $\rightarrow$ dec exp
Apply $\rightarrow$ exp exp
Var $\rightarrow \alpha$
and
Fun $\rightarrow$ *string* β exp
Val $\rightarrow \beta$ exp

Figure 2: Pseudo-CFG productions for types in Fig. 1

$$
\begin{array}{l}
\text{length(x)} = \text{fold}^{list}(\lambda().\text{Zero}, \\
\qquad\qquad\qquad\qquad \lambda(\text{a,r}).\text{Succ(r)}\ )\ \text{x}
\end{array}
$$

Figure 3: Definition of *length* with *fold*

`Nil ->`
$f_{Nil,s_{\uparrow Nil}}$: $\text{s}_{\uparrow Nil} = \text{Zero}$ i.e. $(\lambda().\text{Zero})()$
`Cons -> a list`
$f_{Cons,s_{\uparrow Cons}}$: $\text{s}_{\uparrow Cons} = \text{Succ}(\text{s}_{\uparrow list})$ i.e. $(\lambda(\text{a,r}).\text{Succ(r)})(\text{a},\text{s}_{\uparrow list})$

Figure 4: Definition of *length* with an AG

As we have said in the previous section, the type *list* can be expressed with a pseudo-CFG. Let us briefly recall that an AG is defined over a pseudo-CFG[5] by adding a set of *attributes* on non-terminals (types) and a set of equations over attribute occurrences (*semantic rules*) for each production (constructor function). There are two kinds of attributes: the *synthesized* ones (noted with $\uparrow$) and the *inherited* ones (noted with $\downarrow$). For instance, the function *length* can be defined by the AG specification in Fig. 4, where the unique synthesized attribute is $s_{\uparrow}$. The functions $f_{Nil,s_{\uparrow Nil}}$ and $f_{Cons,s_{\uparrow Cons}}$ are the semantic rules for the *Nil* and *Cons* constructors.

[5]Classically, an AG is defined over a CFG, but here we prefer to define it over a pseudo-CFG and to add a notion of profile of this AG [GGV86, Gie88].

They correspond to the accumulative functions in the fold formalism and, as shown in Fig. 4, they can easily be expressed with (the same) lambda-expressions. For easier comparison, in the sequel, the semantic rules will be given by lambda-expressions.

More generally, an AG specifies, first, a set of synthesized attributes (result variables) and inherited attributes (argument variables) and, secondly, how to *locally* compute the attribute occurrences for each constructor function. More precisely, for each production, each of its *output* attribute occurrences must be defined by a unique semantic rule (equation). The output attribute occurrences for a given constructor function $c \rightarrow t_1 \dots t_n$ are the synthesized attribute occurrences of c (result variables) and the inherited attribute occurrences of each typed variable[6] t_i (argument variables). The other attribute occurrences (inherited of c and synthesized of t_i) are called the *input* attribute occurrences for this constructor. For an inherited attribute occurrence, a semantic rule $f_{c,a_{\downarrow_x}}$ defines the variable $a_{\downarrow_x}$ in production (constructor function) $c \in P$; this variable is the occurrence of inherited attribute $a_\downarrow$ on the non-terminal (type variable) $x \in N$.

In the classical view of AGs, defined over a CFG, the notion of start symbol Z has an important role. In the traditional AG formalism, it is forbidden to declare inherited attributes on the start symbol; this is the main reason for its role. At first sight, this particularity of the AG formalism seems to prevent to bring together AGs and folds. As in [GGV86, Gie88], to remove this constraint, we have introduced in [PDRJ95] the notion of *AG profile* (signature of the function defined by a given AG specification) which replaces the CFG start symbol and is noted $\mathcal{Z}$; all classical attribute evaluation methods can easily be applied to these extended specifications. Since this profile allows to distinguish the argument over which the recursive pattern-matching is applied, which is denoted as the *root argument* of the AG, it is not necessary to give the start symbol in the grammar (notion of pseudo-CFG). Furthermore, it allows to declare additional arguments which are seen, depending on the needs of the application, either as (pre-computed) inherited attributes on the root argument or as global variables. We don't give here the precise definition of an AG profile; it is easy to infer from the examples in [PDRJ95] and those below.

Definition 2.3 (Attribute Grammar) *An* Attribute Grammar *is a tuple* $AG = (G, \mathcal{Z}, A, F)$ *where:*

- $G = (N, T, P)$ *is a pseudo-context-free grammar;*
- $\mathcal{Z}$ *is the* profile *of AG;*
- $A = \bigcup_{X \in N} H(X) \uplus S(X)$ *is a set of attributes, with* $H(X)$ *the* inherited *attributes of* $X \in N$ *and* $S(X)$ *the* synthesized *ones;*
- $F = \bigcup_{c \in P} F(c)$ *is a set of semantic rules, where* $F(c)$ *designates the set of semantic rules of constructor* c*, with* $c \rightarrow t_1 \dots t_n$*. For each synthesized attribute occurrence* $a_{\uparrow_c} \in S(c)$ *(resp. for each inherited attribute occurrence* $a_{\downarrow_{t_i}} \in H(t_i)$*), there exists a semantic rule* $f_{c,a_{\uparrow_c}} \in F(c)$ *(resp.* $f_{c,a_{\downarrow_{t_i}}} \in F(c)$*).*

The main difference between these two syntaxes (*fold* and AG) is that every variable in an AG (attribute occurrence) is explicitly mentioned with a specific name. For example, in the *Cons*-construction of Fig. 4, the expected result over *list* is clearly named $s_{\uparrow_{list}}$ whereas, in the

[6] If the type variable t_i corresponds to a terminal occurrence (α_i), it carries no attribute.

$$\text{append(x,y)} = \text{fold}^{list}(\ \lambda().\text{y},\ \lambda(\text{a,r}).\text{Cons(a,r)}\ )\ \text{x}$$

Figure 5: Definition of *append(x,y)* with *fold*

```
Nil ->
```
$f_{Nil,v_{\uparrow Nil}}$: $\text{v}_{\uparrow Nil} = (\lambda().\text{y})()$
```
Cons -> a list
```
$f_{Cons,v_{\uparrow Cons}}$: $\text{v}_{\uparrow Cons} = (\lambda(\text{a,r}).\text{Cons(a,r)})(\text{a},\text{v}_{\uparrow list})$

Figure 6: Definition of *append(x,y)* with an AG

fold form, this result is implicitly represented by r (called the *accumulative result variable*). The AG form always specifies the name of a variable (attribute occurrence), annotated by the non-terminal it is attached to. This allows to express functions with more than one result (more than one synthesized attribute) and using more parameters (inherited attributes), but we will deal with this point in the next section. In the present section, we will only compare purely-synthesized AGs with first-order folds. Besides, we note that the semantic rules in the AG form are just the accumulating functions of the fold expression applied to explicit parameters (the appropriate attribute occurrences). Note that the profile $\mathcal{Z}$ of the AG for *length* is $length(root : list) \rightarrow s_{\uparrow} : int$.

Another simple example of computation specified with a first-order *fold* is function *append* (Fig. 5), which can be expressed with the purely-synthesized AG of Fig. 6. The profile of this AG is $append(root : list, y : list) \rightarrow v_{\uparrow} : list$, where the first parameter is the root argument and the second parameter is the global variable y.

2.2.2 Functors and Attribute Evaluators

Until now, we have only dealt with notations and expressiveness, but not with the semantics and evaluation of *fold* functions and AGs. The semantics of a function expressed with *fold* is given by the *fold* operator definition, which uses the notion of *functor* [SF93]. The equations in the following definition define the *fold* operator for the simple type *list*.

Definition 2.4 (Fold operator for type *list*) *The* fold^{list} operator *is defined over each constructor of type* list *by:*

$$\begin{array}{lll} fold^{list}(f_n, f_c)\ Nil & = f_n() \\ fold^{list}(f_n, f_c)\ (Cons(a,l)) & = f_c(a, fold^{list}(f_n, f_c)l) \end{array}$$

With this definition, the *length* function in Fig. 3 can be evaluated (computed), since the parameters of f_c (i.e. a and r) are identified. In fact, for the $Cons(a, l)$ constructor, the accumulative result variable r is recursively defined over l. This fold^{list} operator is defined with functors which are statically defined over the *list* type. The way to compute accumulating functions over the structure comes from these functors. They thus give a sense to the evaluation of functions expressed with *fold*.

The semantics of an AG is the solution of the system of equations associated with the set of semantic rules with attribute occurrences over a particular input structure (a tree in the CFG

```
Nil ->
```
$f_{Nil,s_{\uparrow_{Nil}}}$: $s_{\uparrow_{Nil}} = f_n$
```
Cons -> a list
```
$f_{Cons,s_{\uparrow_{Cons}}}$: $s_{\uparrow_{Cons}} = f_c(a,s_{\uparrow_{list}})$

Figure 7: The generic AG for *list*

sense), and this, whatever way is used to resolve this system of equations. In other words, from a given specification (AG), different techniques [Eng84] can be used to generate an attribute evaluator, leading possibly to different evaluations methods but always to the same solution. So, the semantics of an AG is only based on its specification (the signatures of semantic rules) and is independent of the evaluation method. From the AG point of view, the functors associated to the type constructors could be considered as a kind of attribute evaluator.

It is important at this point to note that a set of functors definitions depends only on a given type. They are statically defined from the constructors of this type and are valid for all accumulating functions of all *fold* programs defined on this type.

The equations in Def. 2.4 are the application to the simple type *list* of the more general definition given below. For a given collection of mutually recursive types $T_1, \ldots, T_r$, there is a functor $E_i^{T_s}$ associated with each constructor $C_i^{T_s}$. The goal of this paper is not to precisely redefine this functor (see [SF93]), but informally it describes the way to recursively compute the accumulating functions f_i through the constructor C_i of the recursive structure.

Definition 2.5 (Fold operator) *For a given collection of mutually recursive types $T_1, \ldots, T_r$ and their associated functors $E_i^{T_s}$, the* fold operator *for these types is defined by:*

$$\bigwedge_{s\in[1\ldots r]} fold^{T_s}(\overline{f}) \circ C_i^{T_s} =$$

$$f_i^{T_s} \circ E_i^{T_s}(id_1, \ldots, id_p, fold^{T_1}(\overline{f}), \ldots, fold^{T_r}(\overline{f}))$$

For instance, with type *list*, every computation is performed by recursively applying f_c to the first element of the list and to the forthcoming result over the rest of the list, until f_n can be applied to the *Nil* constructor. This is true, whatever the semantics of f_c and f_n: the *fold* is a *generic* control operator, defined one time for all for a given type.

In the particular case in which an AG represents a *fold* function, the evaluator specified by the functor is a correct attribute evaluator. Furthermore, the class of AGs which correspond to functions expressed with first-order folds is a well-known (actually, the simplest) class of AGs, called *purely-synthesized*, and noted S^1—the "1" refers to the fact that each non-terminal carries a single attribute. Thus, we can define with an AG syntax the generic control operator $\text{fold}^{list}(f_n, f_c)$ over the grammar for *list* (see Fig. 7). Therefore, it is possible to prove that a first-order fold is equivalent to a S^1 AG (see Fig. 8).

2.3 The Normalization Algorithm and Descriptional Composition

Since the expressiveness and the evaluation methods of first-order folds and purely synthesized AGs are very similar, we would like to compare their capabilities to eliminate intermediate data structures. The aim of this section is thus to compare the deforestation methods associated

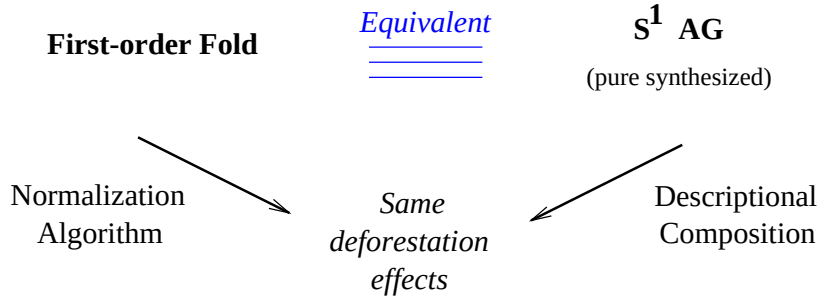

Figure 8: First-order folds and purely-synthesized AGs

Since	length(x) = fold^{list}(λ ().Zero,λ (a,r).Succ(r)) x
and	fold^{list} (f_n, f_c) (Cons(a,l)) = f_c(a,fold^{list}(f_n, f_c) l)
then	length(Cons(a,l)) = Succ(length(l))

Figure 9: *Application to a Construction* over *length*

with both formalisms: the Normalization Algorithm (NA) for first-order folds and Descriptional Composition (DC) for S^1 AGs. We will briefly present each method and its effect on a simple example, the composition *length(append(x,y))*, and then show that they have very similar results (Fig. 8), in spite of perceptibly different approaches.

In the following, we quickly describe the NA method, with its Fold Promotion Theorem, not to give a formal treatment of it, but in order to show that DC is based on a kind of similar but more *symbolic* promotion theorem, in the sense that it is independent of the evaluation method (functor definition).

Normalization Algorithm

The Normalization Algorithm consists of three parts:

- *Generalization* consists in associating some terms with variables, and in replacing such a term by its associated variable each time it is encountered during derivations in NA.
- *Application to a Construction* is the application of the fold operator definition to a constructor and its parameters. For the example of type *list*, the application of the fold^{list} operator over *Cons* with parameters a and l is given by the second equation of Def. 2.4. Fig. 9 presents an example of this step over the *length* function.
- *Fold Promotion* is the fundamental step of NA. It is based on the Fold Promotion Theorem [SF93]. This theorem states that the composition of a function g with a fold function is a new fold function. The new accumulating functions depend on function g and on the accumulating functions of the original fold. For instance, this theorem gives for type *list*:

$$\frac{\begin{array}{rcl}\phi_n() & = & g(f_n()) \\ \phi_c(a, g(r)) & = & g(f_c(a, r))\end{array}}{g(fold^{list}(f_n, f_c)x) = fold^{list}(\phi_n, \phi_c)} \quad (1)$$

The Fold Promotion Theorem ensures the validity of the definition of the resulting fold function when the construction of the ϕ functions is performed locally on each constructor, i.e. each accumulating function in the result depends only on function g and on the accumulating functions of the original fold.

In our previous work [Dur94] comparing Wadler's first deforestation technique [Wad88, Wad90] and DC, we realized that Wadler's algorithm is a more global transformation (i.e. it considers the whole program at once) than DC. In contrast, we will show below that DC has the same local-transformation property as the Fold Promotion Theorem.

It is important to notice, through the following example of *length(append)*, that the role of the Fold Promotion Theorem is to define the ϕ_i accumulating functions and prove the correctness of the resulting fold. The consequence of this transformation is to create new ϕ_i accumulating functions on which the *Application to a Construction* and *Generalization* steps can be applied. More precisely, function g (*length* function in this example) is moved inside the accumulating functions of the resulting fold and hence is directly applied over the original accumulating functions (i.e. their constructor). But the real deforestation process is only performed by the *Application to a Construction* and *Generalization* steps in these new ϕ_i functions.

On the example *length(append)* [SF93], we will explain each step of the NA. First, recall the definitions of each function in fold form:

$$\begin{array}{l} \text{length(x)} = \text{fold}^{list}(\lambda().\text{Zero},\lambda(\text{a,r}).\text{Succ(r)})\ \text{x} \\ \text{append(x,y)} = \text{fold}^{list}(f_n, f_c)\ \text{x} \\ \quad \text{where} \qquad\qquad f_n = \lambda().\text{y} \\ \qquad\qquad\qquad\quad\ f_c = \lambda(\text{a,r}).\text{Cons(a,r)} \end{array}$$

Considering $g = length$, the Fold Promotion Theorem yields:

$$\begin{array}{lll} & \phi_n() & = \text{length}(f_n()) \\ & & = \text{length(y)} \\ \text{and} & & \\ & \phi_c(r_1, r_2) & = \text{length}(f_c(x_1, x_2)) \\ & & = \text{length}(\text{Cons}(x_1, x_2)) \\ & & \qquad \text{with } [x_1/r_1,\ \text{length}(x_2)/r_2] \end{array}$$

Then *Application to a construction* gives (see Fig. 9):

$$\phi_c(r_1, r_2) = \text{Succ}(\text{length}(x_2))$$

And finally, with the *Generalization* of $\text{length}(x_2)$ to r_2:

$$\phi_c(r_1, r_2) = \text{Succ}(r_2)$$

Then, the result of the Normalization Algorithm over *length(append(x,y))* is:

$$\begin{array}{ll} \text{fold}^{list}(& \lambda().\text{fold}^{list}(\lambda().\text{Zero},\lambda(\text{a,r}).\text{Succ(r)})\ \text{y}, \\ & \lambda(\text{r}_1,\text{r}_2).\text{Succ}(\text{r}_2))\ \text{x} \end{array}$$

in which no intermediate structure is produced.

```
Nil ->
    f_{Nil,vs_{↑Nil}}      : vs_{↑Nil} = (λ().length(y)) ()
Cons -> a list
    f_{Cons,vs_{↑Cons}}    : vs_{↑Cons} = (λ(a,r).Succ(r)) (a,vs_{↑list})
```

Figure 10: The DC *(length ∘ append)(x,y)* of *(length(append(x,y)))*

Descriptional Composition

The aim of Descriptional Composition (DC) is to construct, for a given composition of two attribute grammars, $\Omega(G_\Omega) \rightarrow G_\Delta$ and $\Delta(G_\Delta) \rightarrow G_\Theta$, a totally new attribute grammar $(\Delta \circ \Omega)(G_\Omega) \rightarrow G_\Theta$ which has the same semantics as the successive application of Ω and Δ. This new AG does not create the structure corresponding to the intermediate result of type G_Δ (a pseudo-CFG).

Before the definition of DC, let's give the basic idea: the notion of *semantic rule projection*. Intuitively, $(\Delta \circ \Omega)$ is constructed from Ω by replacing each semantic rule whichs computes a term of G_Δ by a projection of semantic rules in Δ over this term. More precisely, if a given semantic rule for the constructor $C_i^{G_\Omega}$ in Ω computes a term t having the type of $C_j^{G_\Delta}$ in G_Δ, then this semantic rule is replaced by the projection of the semantic rules for the constructor $C_j^{G_\Delta}$ in Δ, after the addition of some new attributes to Ω. This projection follows the structure of the original "constructor" semantic rule. DC is a purely syntactic transformation and does not take into account the semantics of the projected semantic rules.[7]

In order to correctly define this semantic rule projection, DC creates a new set of attributes. The names of the new attributes are composed of the name of an attribute of Ω concatenated with the name of an attribute of Δ. To grasp the intuition of this method, we propose to study its effect on our example *length(append)*, presented in Fig. 10.

The following definition of DC is a little restricted, since it forgets both the distinction between semantic and syntactic attributes and the *if–then–else* semantic rule projection. For a complete definition, the reader is refered to the original paper [GG84]. In this definition, we use the type variable $t_{\alpha,0}$ to refer to the type variable C_α in production $C_\alpha \rightarrow t_{\alpha,1} \ldots t_{\alpha,n} \in P_\alpha$, according to the abuse of notation presented in section 2.1.

Definition 2.6 (Descriptional Composition) *Let $G_\Omega = (N_\Omega, T_\Omega, P_\Omega)$, $G_\Delta = (N_\Delta, T_\Delta, T_\Delta)$ and $G_\Theta = (N_\Theta, T_\Theta, P_\Theta)$ be three pseudo-CFGs. Let $\Omega(G_\Omega) \rightarrow G_\Delta$ and $\Delta(G_\Delta) \rightarrow G_\Theta$ be attribute grammars defined by $\Omega = (G_\Omega, \mathcal{Z}_\Omega, A_\Omega, F_\Omega)$ and $\Delta = (T_\Delta, \mathcal{Z}_\Delta, A_\Delta, F_\Delta)$. The* Descriptional Composition *of Ω and Δ generates a new attribute grammar $\Theta(T_\Omega) \rightarrow T_\Theta$, with $\Theta = (\Delta \circ \Omega) = (G_\Omega, \mathcal{Z}_\Omega, A_\Theta, F_\Theta)$ such that:*

1. *for each attribute $a \in A_\Omega(X)$ such that the type of a is $Y \in N_\Delta$, and for each attribute $b \in A_\Delta(Y)$, the attribute ab is declared in $A_\Theta(X)$ with the same type as b;*

2. *for each production $C_\omega \rightarrow t_{\omega,1} \ldots t_{\omega,n} \in P_\Omega$,*
   for each semantic rule $f_{C_\omega, a^0_{x^0}} \in F_\Omega(C_\omega)$ of the form $a^0_{x^0} = C_\delta(a^1_{x^1}, \ldots, a^l_{x^l})$,
   where C_δ is a production in P_Δ of the form $C_\delta \rightarrow t_{\delta,1} \ldots t_{\delta,l}$,
   and for each semantic rule $f_{C_\delta, b^0_{y^0}} \in F_\Delta(C_\delta)$ of the form $b^0_{y^0} = f(b^1_{y^1}, \ldots, b^m_{y^m})$,

[7] Actually, the original definition of DC has a special provision for conditional expressions (*if–then–else*), but here we forget about this possibility.

we define the semantic rule $f_{C_\omega, d^0_{v^0}} \in F_\Theta(C_\omega)$ *with the form* $d^0_{v^0} = f(d^1_{v^1}, \ldots, d^m_{v^m})$
such that, for each $j \in [0..m]$, $v^j = t_{\omega,i}$ *and*
$d^j = a^i b^j$ *if* $y^j = t_{\delta,k}$ *and* $x^k = t_{\omega,i}$ *with* $k \in [0..l]$ *and* $i \in [0..n]$.

For DC to work correctly, the first input AG (Ω) must obey a small restriction: any semantic rule which constructs an input term of the second AG (Δ) must not use a syntactic attribute occurrence more than once (because this would yield a Directed Acyclic Graph instead of a tree).

For the example of $length \circ append$, consider the definition of each function as AGs (Figs. 4 and 6). The semantic rule $f_{Cons,v_{\uparrow_{Cons}}}$ in *append* creates a *Cons*. So, the semantic rule $f_{Cons,s_{\uparrow_{Cons}}}$ of *length* for the *Cons* production is projected onto $f_{Cons,v_{\uparrow_{Cons}}}$, and a new attribute vs is created, leading to the semantic rule $f_{Cons,vs_{\uparrow_{Cons}}}$ in the *Cons* production of $(length \circ append)$ (Fig. 10). Other examples of DC (Fig. 17 for type *list* and Fig. 20 for type *tree*) appear in the next section, which deals with more general AGs with inherited attributes.

Like the difference between fold evaluation and AG evaluation (the functor definition depends only on the type whereas all attribute evaluation methods depend on the form of semantic rules), the difference between NA and DC lies in the knowledge of the evaluation method. DC doesn't require this knowledge at all. In fact, the evaluation method chosen for the result of DC may be different from the evaluation methods of the input AGs.

As said in the introduction, various research works on AGs have exhibited several attribute evaluation techniques applicable to (and actually defining) various subclasses of AGs [Eng84]. The largest one is that of *non-circular* AGs. In [Gie88], the problem of the stability (closure) of an AG class under DC is studied, and it is proved that the non-circular class is stable, i.e. the result of DC on two non-circular AGs is always a non-circular AG.

To interpret the DC method in terms of NA notions, the projection of the semantic rules of DC directly yields the deforested version of the ϕ_i's, whereas those given by the Fold Promotion Theorem must be further deforested by the *Application to a Construction* and *Generalization* steps of NA. Thus, it is possible to see the DC correctness theorem [GG84] as a more symbolic Fold Promotion Theorem, in the sense that the correctness proof for DC is independent of the attribute evaluation method,[8] unlike the Fold Promotion Theorem which is strongly based on the functor definition. Thus, DC is a true source-to-source transformation, independent of any evaluation method: it is a *symbolic composition* without any *Application to a Construction* step.

The *Application to a Construction* step in NA is strongly bound to the evaluation method, i.e. the functor. We see this step as a kind of partial evaluation, which is generalized by the *Generalization* step. If this *Application to a Construction* step is never applied to a bound lambda-term (not a specialization of the input term), then it seems that the first-order fold normalization is equivalent to the descriptional composition of the corresponding S^1 AG (see Fig. 8). In the same way that DC is some kind of symbolic composition, NA can thus be viewed as some kind of generalized partial evaluation.

3 Higher-order Folds and General Attribute Grammars

In this section we present more complex fold program examples, which require to be transformed into second-order fold functions for normalization and which, in the Attribute Grammar

[8] Beware however of DC class-closure problems.

$$\text{reverse(x)} = \text{fold}^{list}(\quad \lambda().\text{Nil},\ \lambda(\text{a,r}).\text{append(r,Cons(a,Nil))}\ )\ \text{x}$$

Figure 11: The "naive" form of the *reverse* fold

```
Nil ->
```
$f_{Nil,s_{\uparrow Nil}}$: $\text{s}_{\uparrow Nil} = (\lambda().\text{Nil})\ ()$
```
Cons -> a list
```
$f_{Cons,s_{\uparrow Cons}}$: $\text{s}_{\uparrow Cons} = (\lambda(\text{a,r}).\text{append(r,Cons(a,Nil))})\ (\text{a,s}_{\uparrow list})$

Figure 12: The AG corresponding to the "naive" form of *reverse*

With the profile $\mathcal{Z} = reverse(root : list, h_{\downarrow} : list) \rightarrow s_{\uparrow} : list$,
and $Id = \lambda(x).x$ being the *identity* function:
```
Nil ->
```
$f_{Nil,s_{\uparrow Nil}}$: $\text{s}_{\uparrow Nil} = Id\ (\text{h}_{\downarrow Nil})$
```
Cons -> a list
```
$f_{Cons,s_{\uparrow Cons}}$: $\text{s}_{\uparrow Cons} = Id\ (\text{s}_{\uparrow list})$
$f_{Cons,h_{\downarrow list}}$: $\text{h}_{\downarrow list} = (\lambda(\text{a,h}).\text{Cons(a,h)})\ (\text{a,h}_{\downarrow Cons})$

Figure 13: The natural AG for *reverse*

context, can be expressed with inherited attributes. As a running example, we will use the deforestation of $(reverse \circ reverse)$ over the *list* type [SF93], which gives the identity (or *copy*) function.

3.1 Informal approach

It is impossible to apply NA on every fold program, just as DC doesn't work on every AG. For example, the *reverse* fold in its *naive* form[9] (Fig. 11) is not *potentially normalizable*, just as DC can't be applied over the corresponding AG (Fig. 12), for the equivalent reason. To solve this problem, Sheard and Fegaras [SF93] introduced the *Second-order Fold Promotion Theorem*.

Since AGs naturally allow the use of inherited attributes, the natural AG form for *reverse* is not the AG of Fig. 12 corresponding to the naive fold form of Fig. 11; rather, it makes use of an inherited attribute, as shown in Fig. 13. DC can then be directly applied on (the composition of) this natural AG (with itself).

In the remainder of this section, we try to compare the second-order (fold) and the inherited (AG) approaches. In spite of their apparent dissimilarities, we notice that there exist some relations between these methods. In fact, the purely-synthesized higher-order AG[10] derived from a given AG with inherited attributes by Knuth's transformation [CM79] is similar to the corresponding second-order fold program.

[9] We say *naive* because, although it is not really natural, this form is the most syntactically simple, basic.

[10] In this paper, the phrase "higher-order AG" does not designate the extension to the AG formalism devised by Vogt *et al.* [VSK89], but rather a (classical) AG with higher-order attributes and semantic rules.

$$\text{reverse(x)} = \text{fold}^{list}(\quad \lambda().\lambda(\text{w}).\text{w},\ \lambda(\text{a,r}).\lambda(\text{w}).\text{r(Cons(a,w))}\ )\ \text{x Nil}$$

Figure 14: The second-order fold form of *reverse*

```
Nil ->
```
$f_{Nil,s_{\uparrow Nil}}$: $\text{s}_{\uparrow Nil} = (\lambda().\lambda(\text{w}).\text{w})$
```
Cons -> a list
```
$f_{Cons,s_{\uparrow Cons}}$: $\text{s}_{\uparrow Cons} = (\lambda(\text{a,s}).\lambda(\text{w}).\text{s(Cons(a,w))})\ (\text{a,s}_{\uparrow list})$

Figure 15: The higher-order AG corresponding to the "natural" AG for *reverse*

3.2 The Higher-order Approach

As we said above, the naive version of the *reverse* fold is not directly normalizable, because the *append* function which it contains (see Fig. 11) works over the accumulative result variable r, which represent the under-construction result of the fold. More precisely [SF93], it is not normalizable because the inner fold is over r, a variable of type *list* bound in the outer fold.

To solve this problem, the transformation $\mathcal{F}_{\mathcal{G}}$ has been introduced as a companion to the second-order Promotion Theorem [SF93]. For the *reverse* example, it automatically translates the naive fold program of Fig. 11 into its second-order equivalent, which is shown in Fig. 14 and amenable to normalization.

Note that the $\mathcal{F}_{\mathcal{G}}$ transformation imposes some restrictions on the base type and the accumulating functions which are used in the naive form. Without giving details (see [SF93]), the type must have a *zero constructor* and the functions used in the input program must be *zero replacement functions*. For instance, in the case of *reverse*, *Nil* is the zero constructor for type *list* and *append* is a zero replacement function. In consequence, the naive form of *reverse* is transformable by $\mathcal{F}_{\mathcal{G}}$ and is then amenable to normalization with the second-order Promotion Theorem.

Since the higher-order *reverse* fold is normalizable, the NA can yield some deforestation results such as *length(reverse(x)) = length(x)* and *reverse(reverse(x)) = copy(x)*.[11]

As the naive version of the *reverse* fold is not directly normalizable, and for the same reasons, DC cannot be applied to the corresponding AG (Fig. 12). Indeed, to apply the DC method, the expressions which construct the intermediate data structure must be directly visible, so that it is possible and correct to project the appropriate semantic rules onto them. In the *reverse* case, the *append* function hides the constructors.

As we said earlier, this problem was not treated in the AG context because, when AGs are written in a natural way, the situation of this *append* is not normally encountered.

However, it is interesting to notice that, in the AG theory, there is a classical transformation which generates an equivalent higher-order synthesized AG from any AG [Knu68, CM79]. For instance, from the *reverse* AG of Fig. 13, which corresponds to the naive *reverse* fold of Fig. 11, this transformation produces the higher-order S^1 AG presented in Fig. 15. The similarity of this higher-order AG with the second-order fold of Fig. 14, produced by the $\mathcal{F}_{\mathcal{G}}$ transformation, is remarkable.

[11] Using the $\mathcal{F}_{\mathcal{G}}^{-1}$ transformation defined in [SF93].

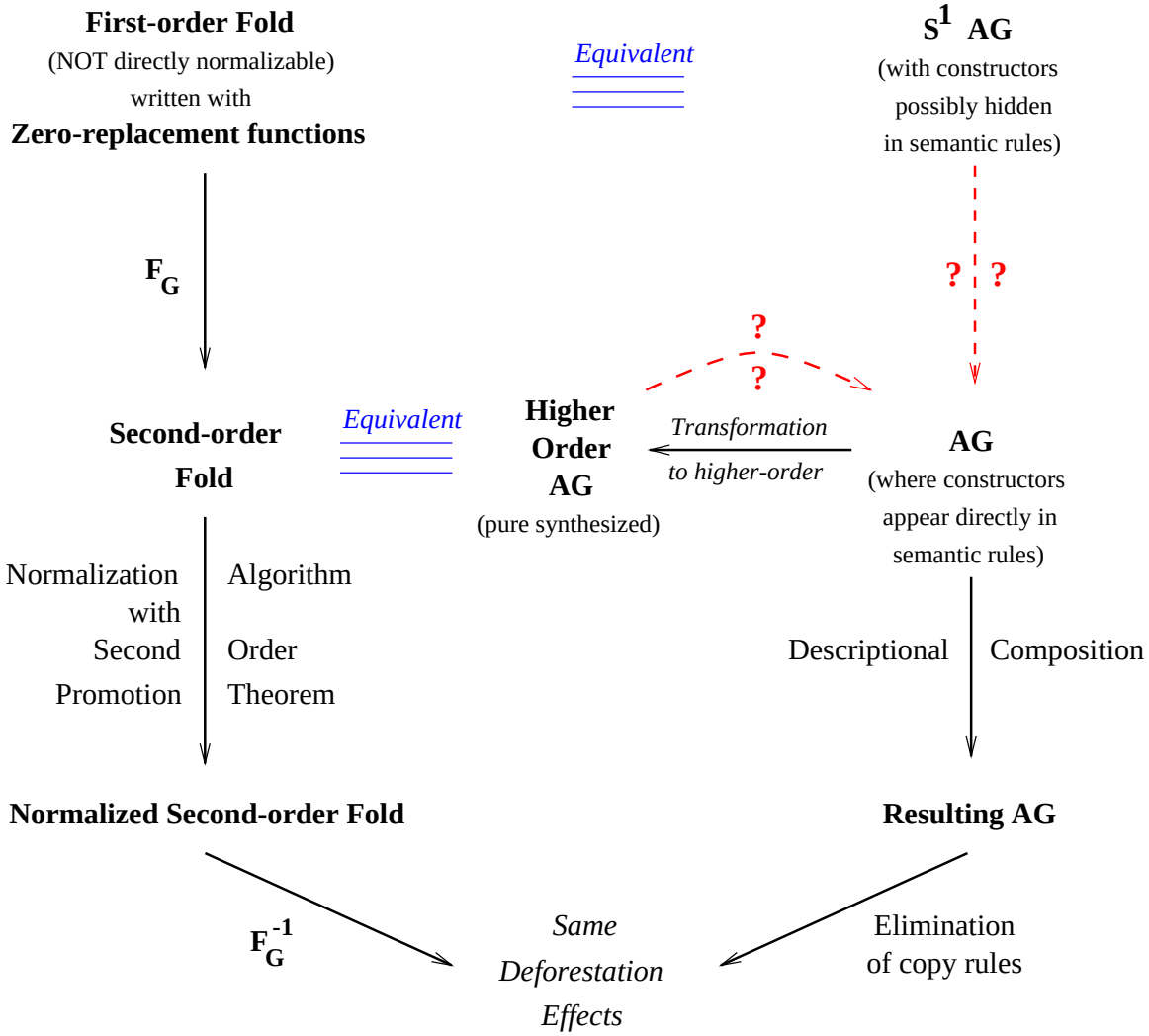

Figure 16: Some relations and transformations between folds and AG

It thus seems interesting to study the possible transposition of the $\mathcal{F}_{\mathcal{G}}$ transformation to the AG formalism. More precisely, in the case of *reverse*, is it possible to automatically transform the naive AG of Fig. 12 into the natural AG of Fig. 13? This hypothetical transformation would be composed of, first, a step similar to the $\mathcal{F}_{\mathcal{G}}$ transformation and, secondly, an "inverse" transformation of the resulting higher-order AG into a first-order AG. These transformations are depicted with dashed arrows in Fig. 16[12]. Even if DC cannot currently process higher-order AGs, such an extension could be inspired from the second-order promotion method with the same restrictions (zero replacement functions).

3.3 The Inherited Approach

Let us recall that the natural programming style, in the AG formalism, uses inherited attributes instead of higher-order semantic rules. In fact, the notions of synthesized and inherited attri-

[12]This figure graphically represents our ideas and possible future work directions, but it should not be considered as giving definite, proven results.

let (ss,hh) = (reverse ∘ reverse) (x,Nil,hh)
in ss

where the profile of (reverse ∘ reverse) is:

$reverse \circ reverse(root : list, sh_{\downarrow} : list, hs_{\downarrow} : list) \rightarrow (ss_{\uparrow} : list, hh_{\uparrow} : list)$

with the following semantic rules:

`Nil ->`

$f_{Nil,hh_{\uparrow Nil}}$: $\mathrm{hh}_{\uparrow Nil} = Id\ (\mathrm{sh}_{\downarrow Nil})$

$f_{Nil,ss_{\uparrow Nil}}$: $\mathrm{ss}_{\uparrow Nil} = Id\ (\mathrm{hs}_{\downarrow Nil})$

`Cons -> a list`

$f_{Cons,sh_{\downarrow list}}$: $\mathrm{sh}_{\downarrow list} = Id\ (\mathrm{sh}_{\downarrow Cons})$

$f_{Cons,hh_{\uparrow Cons}}$: $\mathrm{hh}_{\uparrow Cons} = (\lambda(\mathrm{a,hh}).\mathrm{Cons(a,hh)})\ (\mathrm{a,hh}_{\uparrow list})$

$f_{Cons,hs_{\downarrow list}}$: $\mathrm{hs}_{\downarrow list} = Id\ (\mathrm{hs}_{\downarrow Cons})$

$f_{Cons,ss_{\uparrow Cons}}$: $\mathrm{ss}_{\uparrow Cons} = Id\ (\mathrm{ss}_{\uparrow list})$

Figure 17: The AG resulting from the DC of *reverse* with itself

butes are the basic concept of the AG formalism, introduced right from the beginning [Knu68]. They allow to describe both top-down and bottom-up computations.

Since DC was defined on classical AGs, which use both synthesized and inherited attributes, it works as well on any AG than on S^1 AGs. For instance, DC directly works on the natural AG form of *reverse*, the one in Fig. 13, with inherited attributes.

Now we will present only the result of DC on the *reverse(reverse(x))* example, without giving details on how it is obtained. In fact, with inherited attributes, the basic idea of DC remains the projection of semantic rules.

Fig. 17 presents the AG resulting from the application of DC to *reverse* ∘ *reverse*. In this example, the profile plays an important role. The resulting function takes three arguments: the root argument (the list) and two inherited attributes (*sh* and *hs*). It returns two results which are the two synthesized attributes *ss* and *hh*. The notion of profile is not sufficient to completly define the final AG. In fact, the call to this AG also defines the dependencies between the arguments and the results. For instance, the *hs* argument depends on the *hh* result. Even if this profile notion and this call notation are not classical in the AG formalism, and not formally defined here, we hope that the reader will easily understand what we mean.

Notice that the basic DC transformation leaves many copy rules between attributes, which have no other role than transporting values around the input structure; however, a simple static global analysis can eliminate them in most cases [Rou94]. The result of this elimination on our example of Fig. 17 yields the *copy* AG, which is equivalent to the result obtained by NA.

An example with type *tree*

We will now deal with type *tree*, as defined in Fig. 2, and function *mirror*, which takes a *tree* as argument and returns a *tree* with the same structure but such that, at each node, the two subtrees are inverted. For instance,

mirror (Node (Node (Tip(1), Tip(2)), Tip(3))) = Node (Tip(3), Node (Tip(2), Tip(1)))

`Tip -> i`

$f_{Tip,s_{\uparrow Tip}}$: $\mathrm{s}_{\uparrow Tip} = (\lambda(\mathrm{i}).\mathrm{Tip}(i))\ (\mathrm{i})$

`Node -> left right`

$f_{Node,s_{\uparrow Tip}}$: $\mathrm{s}_{\uparrow Node} = (\lambda(\mathrm{l,r}).\mathrm{Node(r,l)})\ (\mathrm{s}_{\uparrow left},\mathrm{s}_{\uparrow right})$

Figure 18: The AG for *mirror*

let $(r, v) = sigma(t, Zero)$
in r

where the profile of *sigma* is:

$sigma(root : tree, h_{\downarrow} : int) \rightarrow (r_{\uparrow} : tree, v_{\uparrow} : int)$

with the following semantic rules:

`Tip -> i`

$f_{Tip,v_{\uparrow Tip}}$: $\mathrm{v}_{\uparrow Tip} = (\lambda(\mathrm{i,h}).\mathrm{plus(i,h)})\ (\mathrm{i},\mathrm{h}_{\downarrow Tip})$

$f_{Tip,r_{\uparrow Tip}}$: $\mathrm{r}_{\uparrow Tip} = (\lambda(\mathrm{i,h}).\mathrm{Tip(plus(i,h))})\ (\mathrm{i},\mathrm{h}_{\downarrow Tip})$

`Node -> left right`

$f_{Node,h_{\downarrow left}}$: $\mathrm{h}_{\downarrow left} = Id\ (\mathrm{h}_{\downarrow Node})$

$f_{Node,h_{\downarrow right}}$: $\mathrm{h}_{\downarrow right} = Id\ (\mathrm{v}_{\uparrow left})$

$f_{Node,v_{\uparrow Node}}$: $\mathrm{v}_{\uparrow Node} = Id\ (\mathrm{v}_{\uparrow right})$

$f_{Node,r_{\uparrow Node}}$: $\mathrm{r}_{\uparrow Node} = (\lambda(\mathrm{l,r}).\mathrm{Node(l,r)})\ (\mathrm{r}_{\uparrow left},\mathrm{r}_{\uparrow right})$

Figure 19: The AG for *sigma*

`Tip -> i`

$f_{Tip,sv_{\uparrow Tip}}$: $\mathrm{sv}_{\uparrow Tip} = (\lambda(\mathrm{i,h}).\mathrm{plus(i,h)})\ (\mathrm{i},\mathrm{sh}_{\downarrow Tip})$

$f_{Tip,sr_{\uparrow Tip}}$: $\mathrm{sr}_{\uparrow Tip} = (\lambda(\mathrm{i,h}).\mathrm{Tip(plus(i,h))})\ (\mathrm{i},\mathrm{sh}_{\downarrow Tip})$

`Node -> left right`

$f_{Node,sh_{\downarrow right}}$: $\mathrm{sh}_{\downarrow right} = Id\ (\mathrm{sh}_{\downarrow Node})$

$f_{Node,sh_{\downarrow left}}$: $\mathrm{sh}_{\downarrow left} = Id\ (\mathrm{sv}_{\uparrow right})$

$f_{Node,sv_{\uparrow Node}}$: $\mathrm{sv}_{\uparrow Node} = Id\ (\mathrm{sv}_{\uparrow left})$

$f_{Node,sr_{\uparrow Node}}$: $\mathrm{sr}_{\uparrow Node} = (\lambda(\mathrm{l,r}).\mathrm{Node(r,l)})\ (\mathrm{sr}_{\uparrow left},\mathrm{sr}_{\uparrow right})$

Figure 20: The result of DC on *mirror* and *sigma*

Now, consider function *sigma*, which takes as argument a *tree* of integers and returns a *tree* with the same structure, in which the values at the leaves have been added to each other, from left to right. For instance,

sigma (Node (Node (Tip(2), Tip(5)), Tip(3))) = Node (Node (Tip(2), Tip(7)), Tip(10))

These functions can be expressed in the AG formalism, using type *tree* with constructors `Tip` and `Node`. The AG for *mirror* is presented in Fig. 18 and the AG for *sigma* in Fig. 19. We are interested in having a version of function *mirror* $\circ$ *sigma* which doesn't create the intermediate *tree*. The DC of *mirror* and *sigma* yields the AG presented in Fig. 20.

3.4 Comparison and Further Work

In the case of the *reverse* function, we have shown that DC gives a result equivalent to that of NA, without conversion into a higher-order form. The fold formalism cannot express the notion of inherited attribute without resorting to higher-order functions. In fact, the notion of inherited attribute allows to have accumulative variables as arguments of recursive functions. This expressiveness is not currently available in the fold formalism. This is mostly due to the fact that the definition of the fold operator, the generic control operator, is intimately bound to the functor definition. To introduce some inherited notion in the fold formalism, the functor should be defined, not only in terms of the structure of the type, but also in terms of the signatures of the accumulative functions which describe the result and the arguments of the computation. In this context, it would be impossible to statically define the functors using only the type.

One way to define such an *extended functor*, which would take into account both the structure (type) and the form of computations (signatures of *extended accumulative function*), is to try to transpose the techniques for the generation of attribute evaluators. In fact, there exists an "online" translation of AG specifications into functional programs (see [Joh87, EMR93]) but the resulting programs require, in the worst case, a kind of non-strict evaluation called *lenient* [SG95, Tre94]. Lenient evaluation is a sub-class of lazy evaluation which does not allow infinite data structures. As said in the introduction, most researches on AGs have focused on the construction of efficient evaluators, using several techniques which, basically, extract a correct attribute evaluation order from a given AG specification. Hence, regarding lenient evaluation, it is possible to consider these AG evaluation techniques as functional program transformations which transform a lenient program into a strict program [KS86, PDRJ95]. The basic example of this analogy is the circular program "bird" [Bir84, Joh87, PDRJ95]. This transformation, called *lenient decomposition*, decomposes the lenient program into a composition of several strict first-order programs corresponding to the various *visits*. These visits are determined by a static, global analysis of the AG corresponding to the lenient program. Since the NA is more effective on first-order folds than on higher-order ones, this lenient decomposition could be a preliminary step to facilitate the normalization process without resorting to higher order.

Another difference between these two approaches is that, for non-directly-normalizable folds, the NA approach mandates that the input program must be expressed with zero replacement function whereas the DC approach doesn't need this constraint. Even if these particular functions can be automatically derived from the zero constructor of the base type, there exist some types which don't possess such constructors (for instance, type *tree*). In the case of *mirror* $\circ$ *sigma*, the DC method can be directly applied, but NA seems to fail, since we presently don't know how to express *sigma* with zero replacement functions.

Note that, in [FSZ94], this constraint has been relaxed, by systematically passing from a recursive definition to a higher-order fold (*Converting Functional Programs into Algebraic Programs*). But this transformation, which uses an extended normalization algorithm, can still fail in certain cases. Furthermore, the results obtained are in a higher-order form and cannot always be converted down to first order; this is the case of the *reverse* example: the "naive" form with *append* is no more necessary but the "natural" form leads to a second-order fold, unfortunately not convertible to a first-order one. The work of [FSZ94] is an improvement over [SF93], in that it eliminates the constraint on the representation of a program with zero-replacement functions, but some difficulties seem to remain.

With the approach of [FSZ94], the functional version of the *reverse* example (the iterative version) is converted into an inductive version, in higher-order form, on which the NA can be applied in the same way that DC operates on the natural AG form of *reverse*. At this point, we must recall that, in [PRJD96a], we have removed the requirement for the presence of an input tree, i.e. it is really possible to consider the AG formalism as a general-purpose functional declarative language. In our approach, the grammar-based notion of AG is closer to the notion of recursion schemes than to syntax-directed programming. With our extensions, almost every functional program can be converted into an equivalent AG; it is thus possible to argue that the transformations and optimizations made possible by the algebraic programming style in [FSZ94] are also made possible by the AG formalism [vdM94] and its static analysis techniques. But the main difference is that most, if not all, of the research works on AGs avoid to use higher orders. For instance, DC directly works on an AG specification with inherited attributes, without first converting it to a higher-order AG.

4 Conclusion

Even if, to our knowledge, this paper is really the first which compares the fold and attribute grammar formalisms, they reveal many primary similarities.

The first one holds in the expressiveness of the input forms, i.e. a first-order fold program can be expressed with a single-attribute purely-synthesized attribute grammar. In this case, the first-order fold operator definition, based on the functor, yields precisely the attribute evaluation method for this sub-class of attribute grammars. On the other hand, fold evaluation functors are statically deduced from algebraic types whereas most attribute grammar evaluators are statically generated from both algebraic types and semantic rules signatures.

Secondly, in spite of equivalent deforestation results, NA and DC use very different transformation techniques. More precisely, the way to perform the transformation in NA strongly depends on partial evaluation of the input term, whereas DC achieves the deforestation with only a symbolic transformation.

It is clear that this first presentation and comparison of DC and NA remains purely intuitive and informal. We are presently studying how to more precisely formalize these different notations (folds and attribute grammars), definitions (functors and attribute evaluation methods) and transformation methods (NA and DC), with the aim to find possible cross-fertilization opportunities.

From the practical point of view, a prototype implementation of some of these research works has been embedded into our FNC-2 AG system [JPJ+90] (the DC method [RJP94, Rou94, RPJ95] and Dynamic[13] Attribute Grammars [PRJD96b, PRJD96a]).

Our interest for deforestation methods is induced, of course, by the will to eliminate intermediate data structures. In addition, for a couple of years, these methods have been playing an important role in the notion of genericity in the AG formalism [FMY92, BG94, KW94, Rou94, Le 93]. More precisely, an AG on a given grammar (type) defines an algorithm which works only on this type. The notion of genericity is the possibility to reuse (instantiations of) this algorithm on a collection of types more general than only the original one. The method of choice for this notion of instantiation is the composition of AGs [LJPR93, Cor96]. So, the quest for a powerful genericity and modularity framework in the AG programming style is the

[13] see also Conditional Attribute Grammars [Boy96]

main motivation for our current and further research works on the DC method and its possible extensions.

References

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

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

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BG94] John Boyland and Susan L. Graham. Composing tree attributions. In *Conference Record of the 21th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 375–388, January 1994.

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

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

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM Press, 1993.

[CFZ82] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17(2 and 3):163–191 and 235–257, 1982. part I and II See also: rapport 8008, University de Bordeaux I (April 1980).

[CM79] Laurian M. Chirica and David F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979. See also: report TRCS78-2, Dept. of Elec. Eng. and Computer Science, University of California, Santa Barbara, CA (October 1978).

[Cor96] Loïc Correnson. Généricité dans les grammaires attribuées. Rapport de stage d'option, École Polytechnique, 1996.

[DJ90] Pierre Deransart and Martin Jourdan, editors. *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

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

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

[EMR93] Sofoklis G. Efremidis, Khalid A Mughal, and John H. Reppy. AML: Attribute grammars in ML. Tr 93-1401, Cornell University, December 1993.

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

[Far92] Charles Farnum. Pattern-based tree attribution. In *Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 211–222, Albuquerque, New Mexico, January 19–22 1992.

[Feg96] Leonidas Fegaras. Fusion for Free! Technical Report 96-001, Oregon Graduate Institute of Science and Technology, Portland, January 1996.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Progr. Languages*, pages 223–234. ACM press, Albuquerque, NM, January 1992.

[FSS92] Leonidas Fegaras, Tim Sheard, and David Stemple. Uniform traversal combinators: Definition, use and properties. In *11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lect. Notes in Comp. Sci.*, pages 148–162, Saratoga Springs, New York, June 1992. Springer-Verlag.

[FSZ94] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Symp. on Compiler Construction*, volume 19(6), pages 157–170. *ACM SIGPLAN Notices*, Montréal, June 1984.

[GGV86] Harald Ganzinger, Robert Giegerich, and Martin Vach. MARVIN: a tool for applicative and modular compiler specifications. Forschungsbericht 220, Fachbereich Informatik, University Dortmund, July 1986.

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

[GLJ93] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Func. Prog. Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, New York–Heidelberg–Berlin, September 1987. Portland.

[JP] Martin Jourdan and Didier Parigot. A bibliography on attribute grammars. `http://www-rocq.inria.fr/oscar/FNC-2/AG.html` Updated regularly. Contains around 850 references to papers on Attribute Grammars. INRIA, France.

[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 *Conf. on Programming Languages Design and Implementation*, volume 25(6), pages 209–222. *ACM SIGPLAN Notices*, White Plains, NY, June 1990.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[KS86] Matthijs F. Kuiper and S. Doaitse Swierstra. Using Attribute Grammars to Derive Efficient Functional Programs. Report RUU–CS–86–16, Utrecht University, 1986.

[KW94] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. In *Acta Informatica*, volume 31, pages 601–627, 1994.

[Le 93] Carole Le Bellec. *La généricité et les grammaires attribuées.* PhD thesis, Département de Mathématiques et d'Informatique, Université d'Orléans, 1993.

[LJPR93] 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.

[MFP91] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf on Func. Prog. Languages and Computer Architecture*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144. Springer-Verlag, 1991.

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

[PDRJ95] 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`.

[PDRJ96] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées: un langage fonctionnel déclaratif. In *Journées Francophones des Langages Applicatifs*, Val-Morin, Québec, January 1996. `ftp://ftp.inria.fr/INRIA/Projects/ChLoE/FNC-2/publications/gdr.ps.gz`.

[PRJD96a] 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`.

[PRJD96b] Didier Parigot, Gilles Roussel, Martin Jourdan, and Étienne Duris. Dynamic Attribute Grammars. In *International Symposium on Programming Languages, Implementations, Logics and Programs*, Lect. Notes in Comp. Sci., Aachen, September 1996.

[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.

[RPJ95] Gilles Roussel, Didier Parigot, and Martin Jourdan. Static and Dynamic Coupling Attribute Evaluators. Rapport de recherche 2670, INRIA, October 1995. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2670.ps.gz`.

[Ses] Peter Sestoft. A bibliography on partial evaluation. `ftp://ftp.diku.dk/pub/diku/dists/jones-book/partial-eval.bib.Z` Updated regularly. Contains around 500 references to papers on partial evaluation. DIKU, University of Copenhagen, Denmark.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press.

[SG95] Klaus E. Schauser and Seth C. Goldstein. How much non-strictness do lenient programs require? In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 216–225, La Jolla, CA, USA, June 1995. ACM Press.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 306–313, La Jolla, CA, USA, 1995. ACM Press.

[Tre94] Guy Tremblay. *Parallel implementation of lazy functional languages using abstract demand propagation.* Phd thesis, McGill University, Montreal Canada, November 1994.

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

[VM82] Aare O. Vooglaid and Merik B. Méristé. Abstract attribute grammars. *Progr. and Computer Software*, 8(5):242–251, September 1982.

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *Conf. on Progr. Languages Design and Implementation*, volume 24(7), pages 131–145. *ACM SIGPLAN Notices*, Portland, OR, July 1989.

[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.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science*, volume 73, pages 231–248, 1990. (Special issue of selected papers from 2'nd European Symposium on Programming).

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, 655, avenue de l' Europe, 38330 MONTBONNOT ST MARTIN
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