Attribute Grammars and Functional Programming Deforestation

Loïc Correnson, Etienne Duris, Didier Parigot, Gilles Roussel

INRIA,
Domaine de Voluceau,
Rocquencourt, BP 105,
78153 Le Chesnay Cedex, France.
E-mail: `Firstname.Name@inria.fr`
Web pages: `http://www-rocq.inria.fr/oscar/FNC-2/`
Tel: (+33) 1 39 63 55 46 — Fax: (+33) 1 39 63 58 84

Gilles Roussel is also with
Université de Marne-la-Vallée,
2, allée du Promontoire,
93166 Noisy-le-Grand, France.
E-mail: `roussel@univ-mlv.fr`

Abstract

The functional programming community is paying increasing attention to static structure-based transformations. For example, generic control operators, such as *fold*, have been introduced in functional programming to increase the power and applicability of a particular kind of static transformation, called *deforestation*, which prevents the construction of useless intermediate data structures in function composition. This is achieved by making the structure of the data more explicit in program specifications.

We argue that one of the original concepts of Attribute Grammars is precisely to make data structures explicit in program specifications. Furthermore, there exists a powerful static deforestation-like transformation in their context. In this paper, we present similarities between deforestation methods, on the one hand with the functional approach, and on the other hand with the Attribute Grammars approach.

In order to gain a grasp of these similarities, we first make a simple comparison: purely-synthesized Attribute Grammars and first order folds. In this context, deforestation transformations are equivalent. This allows us to highlight the limitations of the fold formalism and to present how the hylomorphism approach generalizes it; hylomorphisms and attribute grammars are surprisingly alike. Finally, we show how the inherited attribute notion in Attribute Grammars solves some transformation problems in higher order functional programs.

Keywords: Attribute grammars, functional programming, deforestation, structure-directed programming, static analysis, partial evaluation.

1 Introduction

The functional programming community is paying increasing attention to structure-based transformations. To make these transformations simpler, data structures in programs should be explicit. We argue that one of the original concepts of Attribute Grammars (AGs) is precisely to make data structures explicit in program specifications. Here, we point out the similarities and relations between AGs and functional programming paradigms, especially in the context of a data structure-based transformation, called *deforestation*, which prevents the construction of useless intermediate data structures in function composition.

An AG is a declarative specification of computations over structures [Knu68, AM91, Paa95]. Commonly these structures are concrete, but Dynamic AGs [PDRJ95, PRJD96] work equally well with abstract structures, like a computation recursion scheme. We could then consider an AG as a function with an argument driving the calculus, possibly with other arguments as inherited attributes (computed top-down), and which returns its result as a synthesized attribute (computed bottom-up). This AGs functional view has been already studied, for example in [GG84, Joh87, PDRJ96].

Nevertheless, AGs specifications have some important characteristics which allow them to be clear, concise and easily maintained [JPJ+90, JP92, Paa95]. First they are *declarative*, i.e., program specification is completely independent of any evaluation order [Eng84]. Secondly they allow a complex computation to be decomposed into small easily understandable local parts (on each constructor), which are joined together by the generator evaluator. Finally they make the structure (constructors) explicit in the specification, facilitating structure-based transformations (like deforestation) or *structure-directed genericity* [DPRJ97] (i.e., instantiation of an algorithm for a new structure, giving a relation between old and new structures).

From a given structure T_1, the successive application of two AGs $f : T_1 \rightarrow T_2$ and $g : T_2 \rightarrow T_3$ yields a structure of type T_3. In such a case, the intermediate structure of type T_2 is useless, and the aim of the *Descriptional Composition* (DC) [GG84, GGV86, Gie88, FMY92, Rou94] is to statically transform the f and g composition into a new AG specification which no longer uses T_2-constructors. The basic idea for the DC is to recognize in f the use of a constructor c and plunge (integrate) the computation associated with c in g into this "use" (semantic rule projection).

This intermediate data structure elimination is also the goal of functional *deforestation*. Most deforestation techniques try to exploit a kind of generic control operator to capture both the pattern of recursion of a function and the pattern of recursion of the type definition. The *shortcut deforestation* of Gill, Launchbury and Peyton Jones [GLJ93] makes it possible for lists, using a `foldr/build` elimination rule. To take every type into account, Sheard and Fegaras [SF93, FSZ94, LS95] consider in their *Normalization Algorithm* (NA) the fold operator (a *catamorphism*) related to a *functor* automatically generated from algebraic type definition. Thanks to Takkano and Meijer's studies [TM95, MFP91], Hu, Iwasaki, Takeishi and Onoue [HIT96, OHIT97] have recently generalized these ideas and homogenized these notions with *hylomorphisms*. This has lead to the automatic exploitation of constructive deforestation theorems like *Acid Rain* [TM95] in the HYLO system [OHIT97], which uses a static transformation from functional programs into hylomorphisms.

All these deforestation in *calculational form* methods, sometimes also called *fusion* methods are based on functors, catamorphisms, hylomorphisms and more generally on *Constructive Algorithmics* [MFP91]. They are similar to the AGs DC in that they are based on local transformations (at constructor level). In our previous work [Dur94] comparing Wadler's first deforestation technique [Wad88] and the DC, we realized that Wadler's algorithm is a more global transformation than the DC (i.e. it considers the whole program at once). In contrast to this method, we will show that the DC has the same local-transformation property as the *Fold Promotion Theorem*, which is the basis of the NA [SF93]. Furthermore, the NA and the DC provide the same results for first order functions. In spite of these similarities, a particularity of DC is that it is a source to source transformation, completely independent of any evaluation order, whereas in fusion methods, the functor (type) gives the function evaluation scheme: fusion could be viewed as a generalized partial evaluation.

Moreover, for more complex programs, AGs naturally use inherited attributes rather than higher order functions. From the beginning of AGs, inherited attributes have been common objects among other attributes (unlike higher order functions among first order ones in functional programming). Thus, for some complex specifications, fusion transformation needs special treatments due to higher order functions where the DC only has to deal with inherited attributes. For example, the well-known *reverse* function (for lists) composed twice leads to a simple "copy" AG by the DC, whereas this is not necessarily so with fusion methods.

The remainder of this article is divided into two sections. The first one presents and compares fold formalism and its NA with AGs and its DC in the simple context of first-order functions. The second section highlights the difficulties of less simple cases; it presents existing solutions and extends the comparisons to the HYLO formalism.

2 Folds and AGs in Easy Cases

In this section, we present a comparison between AG and a particular functional programming style, the fold formalism, especially introduced for deforestation algorithms. Firstly, we intentionally choose to deal only with this fold notation in order to gain a grasp of similarities, and to avoid the complexity inherent in multiplicity of different formalisms, notations, and vocabulary. The advantage is to allow the reader to compare underlying ideas and methods rather than formalisms and techniques, which can be seen as particular. We will see in the next section that this choice is not all that restrictive. So, by "simple cases", we mean classical first-order functional programs, which correspond to purely-synthesized AGs.

2.1 Notations

In [SF93], fold generic control operators are defined over mutually recursive sum-of-product types (algebraic type definitions), which show explicitly the type constructors. Historically, AGs are based on the *Context-Free Grammar* notion, but can also be defined on such algebraic type definitions [CM79, GG84] in which the type constructors play the role of productions.

Over the simple recursive type *list*, the classical function *length* can be defined using the generic control operator fold for this type (Figure 1). The two lambda-expressions in this definition 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. 2.1.

An AG specification, over an algebraic type and a given set of attributes, is a set of equations over attribute occurrences for each type constructor (production). There are two kinds of attributes: *synthesized* (noted with $_\uparrow$) and *inherited* (noted with $_\downarrow$). For instance, function *length* can be defined by the AG specification in Figure 2, where the single attribute $s_\uparrow$ is synthesized. The functions $f_{Nil,s_{\uparrow_{Nil}}}$

$$length\ (x) = fold^{list}\ (\lambda().Zero,$$
$$\lambda(a,r).Succ(r)\ )\ x$$

Figure 1: Definition of *length* with *fold*^{list}

$length\ (root : list) \rightarrow s_{\uparrow} : list$
`Nil ->`
$f_{Nil,s_{\uparrow_{Nil}}} \quad : \ s_{\uparrow_{Nil}} = (\lambda().Zero)()$
`Cons -> a list`
$f_{Cons,s_{\uparrow_{Cons}}} \quad : \ s_{\uparrow_{Cons}} = (\lambda(a,r).Succ(r)\ )\ (a, s_{\uparrow_{list}})$

Figure 2: Definition of *length* with an AG

and $f_{Cons,s_{\uparrow_{Cons}}}$ are the semantic rules for the *Nil* and *Cons* constructors. They correspond exactly to the accumulative functions in the fold formalism and, as shown in Figure 2, they can be easily expressed with (the same) lambda-expressions. Then the AG for *length* could be viewed as a function, whose signature is the *profile* of the AG [Gie88], i.e., $length(root : list) \rightarrow s_{\uparrow} : list$. This means that the input parameter, called *root*, is of type *list*, as is the result, which is computed in the synthesized attribute $s_{\uparrow}$.

The main difference between these two notations (fold and AG) is that every attribute occurrence in an AG is explicitly mentioned with a specific name. For example, in the *Cons*-construction of Figure 2, the expected result over *list* is clearly named $s_{\uparrow_{list}}$ whereas, in the 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 functions to be expressed with more than one result (more than one synthesized attribute) and using more parameters (inherited attributes).

2.2 Functors and Attribute Evaluators

After establishing the similarity of notations and expressiveness of fold functions and AGs, we now deal with their semantics and evaluation.

The semantics of a function expressed with fold is given by the fold operator definition, which uses the notion of *functor* [SF93]. The following equations define the fold operator for the simple type *list*.

Definition 2.1 (Fold operator for type *list* [SF93]) *The fold*^{list} *operator is defined for each constructor of type* list *by:*

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

With this definition, the *length* function in Figure 1 can be evaluated (computed), since the parameters of f_c (i.e., a and r) are identified. As can be seen in Def. 2.1, 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 and automatically defined over the *list* type constructors. These functors show how to compute accumulating functions over the structure. They thus give a meaning to the evaluation of functions expressed with fold. Note that the recursion scheme of the computation is strictly identical to that of the type definition: this is the essence of structure-directed programming.

$$
\begin{aligned}
&append\ (x,y) = fold^{list}\ (f_n, f_c)\ x \\
&\quad \text{where} \qquad f_n = \lambda().y \\
&\qquad\qquad\qquad\ \ f_c = \lambda(a,r).Cons(a,r)
\end{aligned}
$$

Figure 3: Definition of *append(x,y)* with $fold^{list}$

$append\ (root : list, y : list) \rightarrow v_{\uparrow} : list$

`Nil ->`

$f_{Nil, v_{\uparrow_{Nil}}} \quad : \ v_{\uparrow_{Nil}} = (\lambda().y)\ ()$

`Cons -> a list`

$f_{Cons, v_{\uparrow_{Cons}}} \quad : \ v_{\uparrow_{Cons}} = (\lambda(a,r).Cons(a,r))\ (a, v_{\uparrow_{list}})$

Figure 4: Definition of *append(x,y)* with an AG

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 sense), whichever method is used to solve this system of equations. In other words, from a given specification (AG), different techniques can be used to generate an attribute evaluator, leading possibly to different evaluation methods but always to the same semantics.

When an AG represents a fold function, the evaluator specified by the functors is a correct but particular attribute evaluator. Indeed, functor definitions depend only on the given type(s). They are statically derived from the constructors of this type and are valid for all accumulating functions of all fold programs defined on this type. Furthermore, the class of AGs which corresponds to functions expressed with folds is a well-known (actually, the simplest) class of AGs, called *purely-synthesized* [Eng84], and noted S^1—the "1" refers to the fact that each non-terminal carries a single attribute (see also a translation from AGs to catamorphisms in [FJMM91]).

2.3 Deforestation

Since the folds and purely-synthesized AGs expressiveness and evaluation methods are very similar, the aim of this section is to compare the static deforestation methods associated with both formalisms: the Normalization Algorithm for first-order folds and the Descriptional Composition for S^1 AGs. In the following, we briefly describe each method, not to give a formal treatment of them, but in order to show that the DC is based on a similar but more *symbolic* Promotion Theorem, in the sense that it is independent of the evaluation method (functor definition). We will illustrate their effects on the simple example of *length(append)* [SF93]. Figure 3 shows the fold function for *append*, and Figure 4 shows the corresponding AG.

Normalization Algorithm

For a given function in its fold definition, the NA allows the composition of another function g with this fold to be deforested, integrating g in the the fold accumulating functions, and avoiding intermediate structure constructions.

The NA comprises three parts [SF93]:

- *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 the NA.

$$
\begin{array}{lll}
\text{Since} & length\ (x) & = fold^{list}\ (\lambda().Zero, \lambda(a,r).Succ(r)\ )\ x \\
\text{and} & fold^{list}\ (f_n, f_c)\ (Cons(a,l)) & = f_c\ (a, fold^{list}\ (f_n, f_c)\ l) \\
\text{then} & length\ (Cons(a,l)) & = Succ(length\ (l))
\end{array}
$$

Figure 5: *Application to a Construction* on *length*

$$
\begin{array}{l}
fold^{list}\ (\lambda().length\ (y), \\
\quad \lambda(r_1, r_2).Succ(r_2)\ )\ x
\end{array}
$$

Figure 6: Result of the NA for *length(append(x,y))*

- *Application to a Construction* is the application of the fold operator definition to a constructor and its parameters. Figure 5 presents an example of this step on the *length* function.
- *Fold Promotion* is the fundamental step of the NA. It is based on the Fold Promotion Theorem[1] [SF93]. This theorem states that the composition of a function g with a fold function is a new fold function. For instance, this theorem gives for type *list*:

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

The Fold Promotion Theorem ensures the validity of the resulting fold function definition, when the construction of the ϕ functions is performed locally on each constructor, i.e., each new accumulating function in the result depends only on function g and on the accumulating functions of the original fold. The main role of the Fold Promotion Theorem is to define the ϕ_i accumulating functions of the resulting fold, on which the *Application to a Construction* and *Generalization* steps can be applied. More precisely, function g is moved inside the accumulating functions of the resulting fold and hence is directly applied over the original accumulating functions (i.e., their constructors). But the real "deforestation" process (i.e., eliminating structure constructors) is only performed by the *Application to a Construction* and *Generalization* steps in these new ϕ_i functions.

The result of the NA on *length(append)*, which no longer contains any *Cons* constructors, is given in Figure 6.

Descriptional Composition

The aim of the DC is to staticaly construct, for a given composition of two attribute grammars $\Omega(T_1) \rightarrow T_2$ and $\Delta(T_2) \rightarrow T_3$, a new attribute grammar $(\Delta \circ \Omega)(T_1) \rightarrow T_3$ which has the same semantics as the successive application of Ω and Δ. This new AG will not, however, create the structure corresponding to the intermediate result of type T_2.

The basic idea is the notion of *semantic rule projection.* Intuitively, $(\Delta \circ \Omega)$ is constructed from Ω by replacing each semantic rule which computes a term of T_2 by a projection of the semantic rules in Δ over this term. More precisely, let f_i be a semantic rule for the constructor $C_i^{T_1}$ in Ω, which computes a term t of type $C_j^{T_2}$ ($C_j^{T_2}$ is a constructor in Δ; let $\overline{g}$ be its associated semantic rules). Then, in $(\Delta \circ \Omega)$,

[1] This theorem is an instance of a famous law in category theory, often called *fixed point fusion* [MFP91].

$$
\begin{array}{l}
(length \circ append)\ (root : list, y : list) \rightarrow vs_{\uparrow} : list \\
\texttt{Nil ->} \\
\quad f_{Nil,vs_{\uparrow_{Nil}}} \ : \ vs_{\uparrow_{Nil}} = (\lambda().length\ (y))\ () \\
\texttt{Cons -> a list} \\
\quad f_{Cons,vs_{\uparrow_{Cons}}} \ : \ vs_{\uparrow_{Cons}} = (\lambda(a,r).Succ(r))\ (a, vs_{\uparrow_{list}})
\end{array}
$$

Figure 7: The DC *(length ∘ append)(x,y)* of *(length(append(x,y)))*

---

f_i is replaced by the projection of $\overline{g}$ onto f_i. This projection follows the structure of the original "constructor" semantic rule f_i. It also creates new attributes: for each attribute a of X in Ω such that the type of a is a non-terminal Y of Δ, we declare for each attribute b of Y in Δ, an attribute ab on X in Θ. The type of a new attribute ab is the type of b.

The DC is a purely syntactic transformation. It neither takes into account the semantics of the projected semantic rules, neither their evaluation order. This presentation of the 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 formal definition, the reader should refer to [GG84, Rou94].

Since the AG definitions for *length* and *append* appear in Figure 2 and 4, the effect of the DC on the composition *length(append)* is presented in Figure 7. For instance, 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 $(length \circ append)$.

Comparisons and Discussion

We have seen in previous sections the difference between folds and AGs evaluation: the functor definition depends only on the type whereas all attribute evaluation methods depend on the form of semantic rules. In the same way, the difference between the NA and the DC depends on the knowledge of the evaluation method. The DC doesn't require this knowledge at all. In fact, the chosen evaluation method for the DC result may differ from the evaluation methods of the input AGs.

To interpret the DC method in terms of the NA notions, the projection of the semantic rules of the DC directly yields the deforested version of the ϕ_i's, whereas in the NA, ϕ_i's given by the Fold Promotion Theorem must be further deforested by the *Application to a Construction* and *Generalization* steps. Thus, it is possible to consider the DC correctness theorem [GG84] as a more symbolic Fold Promotion Theorem. By this, we mean that the correctness proof for the DC is independent of the attribute evaluation method, unlike the Fold Promotion Theorem which is strongly based on the functor definitions.

Various research studies 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 the DC is studied, and it is proved that the non-circular class is stable, i.e., the DC result on two non-circular AGs is always a non-circular AG. Thus, the 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 the NA is strongly tied to the evaluation method, i.e., the functor. We view this step as a kind of partial evaluation, which is generalized by the *Generalization* step. Thus, in spite of equivalent result of first-order fold normalization and the DC of the corresponding S^1 AG, their basic principle are slightly different. In the same way that the DC is a kind of symbolic composition, the NA can thus be viewed as a kind of generalized partial evaluation.

3 More General Cases

The fold formalism is based on the idea of representing in a uniform way both the pattern of recursion of a function and the pattern of recursion of the underlying type definition. This idea is supported by the more general concept of catamorphism. For a given algebraic type definition, a functor can be determined [SF93], which represents its recursion scheme. Then, a function expressed relating to this functor could be easily applied over a structure of the given type, replacing each constructor by the corresponding part of the function definition. Many studies concerning deforestation have chosen this way to make explicit type constructors, facilitating their detection and, when possible, their elimination in function composition.

In this section, we highlight some remaining problems in fold formalism discussed in the "simple cases" of the previous section. For each of these, we present related solutions provided by fold itself, but also by other calculational approaches. To avoid introducing multiple formalisms in addition to fold, we will only deal with the HYLO approach [TM95, HIT96, OHIT97], but the reader may find more details in [MFP91, Fok95, TM95, HIT96]. However, this formalism seems to be sufficiently general and homogeneous for the necessity of our discourse. We also present the AGs approaches related to these points, and finally, we present higher order functional programming versus inherited attributes of AGs in the deforestation problem.

First, we enumerate three important remarks about the original fold formalism:

1. Using fold [FSZ94], it is possible to express many complex types or calculation schemes. However, each new concept extending the fold expressive power induced complex modifications of the transformation algorithms: thus, this fusion is not easily "maintainable".

2. The NA is a complex rewriting system which is not easy to automate, and which does not necessarily lead to a deforested form (see exceptions in [FSZ94]).

3. To be deforested, a program must be expressed using the specific fold formalism: this seems to be exacting in practical functional programming.

3.1 Hylomorphisms

To improve formalisms and transformations relating to these three points, deforestation in calculation form methods use categorical notions. We will just give here a short recall of basic notions and present it with the simple type *list* with elements of type A. The underlying idea is the following: a data type is a collection of operations (constructors) denoting how to construct each element of this type. These operations allow functions to be defined on this type. So a data type is a particular F-algebra, where F is a functor from a category $\mathcal{C}$ to $\mathcal{C}$. When F is polynomial, i.e., built up by the four basic functors Identity (Id), Constant ($!A$), Product ($\times$) and Separated Sum ($+$), the category of F-algebras has an initial algebra, noted μF, which is defined according to the type and its constructors [MFP91, TM95, HIT96].

For example, the type $ListA$

$$ListA = Nil \mid Cons(A, ListA)$$

is categorically defined by the initial object

$$\mu L = (ListA, in_L)$$

where L is the functor

$$L = !1 + !A \times Id$$

and in_L is the *data constructor*

$$Nil \triangledown Cons$$

where $\triangledown$ is the operation related to the separated sum: $(f \triangledown g)(1,x) = fx$ and $(f \triangledown g)(2,y) = gy$.

The *data destructor* out_L is the inverse operation to in_F :

$$\begin{array}{lll} out_L & = \lambda xs. \text{ case } xs \text{ of} & \\ & \quad Nil \rightarrow (1,()) & \\ & \quad Cons(a,as) \rightarrow (2,(a,as)) & \end{array}$$

In the same way, the natural integer type Nat is defined by $\mu N = (Nat, in_N)$ where $N = !1 + Id$ and $in_N = Zero \triangledown Succ$.

Hylomorphisms [MFP91] represent both catamorphisms (data structure consumers like fold) and anamorphisms (data structure producers like the dual of fold). Represented in a homogeneous way, *hylomorphisms in triplet form* [TM95] allow Onoue, Hu, Iwasaki and Takeichi's HYLO system [HIT96, OHIT97] to improve the deforestation method with respect to the three remarks given above:

1. Hylomorphisms in triplet form cover all those in [SF93, FSZ94, TM95]. The idea is to distinguish three steps in such a hylomorphism: a step to "abstract" the input type in an algebra, another step to construct the output type from an abstracted form and, between them, the real transformation which is performed on the type abstractions. For example, the hylomorphism in triplet form representing the function *length* is defined as follows:

$$\begin{array}{lll} length & : & ListA \rightarrow Nat \\ & = & [\![\phi,\eta,\psi]\!]_{N,L} = [\![in_N, Id + \pi_2, out_L]\!]_{N,L} \end{array}$$

   In this definition, $\psi = out_L$ abstracts the input list in a L-algebra; $\eta = Id + \pi_2$ transforms this L-algebra abstraction in a N-algebra abstraction, ignoring by π_2 (the second projection on a pair) the information related to list elements, which is useless in the length computation; finally, $\phi = in_N$ constructs, from the N-algebra abstraction, the output concrete natural. This decomposed abstract specification on polynomial functors allows the deforestation algorithm to be more homogeneous, more general and easier to maintain. Notice that both catamorphism $(\![\phi]\!)_F = [\![\phi, id, out_F]\!]_{F,F}$ and anamorphism $[\!(\psi)\!]_F = [\![in_F, id, \psi]\!]_{F,F}$ are particular cases of hylomorphisms.

2. The Fold Promotion Theorem has its equivalent fusion law in this formalism [HIT96]:

   Theorem 3.1 (Hylo Fusion) *The Left Fusion law of a function with an hylomorphism (there is also the symmetrical Right Fusion law) is:*

$$\frac{f \circ \phi = \phi' \circ Gf}{f \circ [\![\phi,\eta,\psi]\!]_{G,F} = [\![\phi',\eta,\psi]\!]_{G,F}}$$

   A nice property verified by hylomorphisms in triplet form is:

   Theorem 3.2 (Hylo Shift)

$$[\![\phi,\eta,\psi]\!]_{G,F} = [\![\phi\circ\eta, id, \psi]\!]_{F,F} = [\![\phi, id, \eta\circ\psi]\!]_{G,G}$$

This allows the fusion laws to be specialized in the *Acid Rain Theorem* [TM95], which is easier to apply in a constructive algorithm.

Theorem 3.3 (Acid Rain) *The Cata-Hylo fusion law (there is also the symmetrical Hylo-Ana law) is:*

$$\frac{\tau : \forall A.(FA \to A) \to F'A \to A}{[\![\phi, \eta_1, out_F]\!]_{G,F} \circ [\![\tau in_F, \eta_2, \psi]\!]_{F',L} = [\![\tau(\phi \circ \eta_1), \eta_2, \psi]\!]_{F',L}}$$

The "fusion step", which consists in deducing both ϕ_i from g and ϕ in Fold Promotion, and ϕ' from f and ϕ in Hylo-Fusion, is difficult to automate: in the Acid Rain case, this problem is moved into the "hylomorphism restructuring phase" of the HYLO system (we give some explanations in the next section, second point).

3. There exists an automatic transformation [HIT96, OHIT97] from classical functional programs into their HYLO forms. This transformation saves the programmer from using a non trivial formalism (hylomorphisms) in a program specification.

3.2 Attribute Grammars

In this section, we explain, for each enumerated remark, why the AGs approach gives, from our point of view, a worthwhile solution.

1. Like in calculational formalisms such as HYLO, there exists in the AGs theory an abstracted notion for types, provided by the context-free grammar notation (BNF). In fact, all basic AGs transformation algorithms use this abstraction. Moreover, the DC is a purely syntactic transformation, independent of any functor expressed on this abstract type. The DC algorithm is fully generic, i.e., independent of any type representation (algebra) and any evaluation order (functor).

2. Since the DC algorithm is only based on the *semantic rule projection* step, which is purely syntactic, it can be applied without any modification for more complex types. Moreover, the DC works only at the specification level, which corresponds to the η part in the triplet form, when ϕ and ψ "just" contain the data constructor and destructor. This is not always the case with the HYLO method. In fact, to make Hylo-Fusion more efficient, the ϕ (resp. ψ) part of the hylomorphism in triplet form should contain as much computation as possible. For the Acid Rain Theorem, the ϕ and ψ parts must be simple, not so much for deforestation power, but rather to ease the recognition of in_F (resp. out_F) in ϕ (resp. ψ). So, it seems that the constructive feature of the Acid Rain Theorem is not so adequate for the large expressive power of triplet form notation. On the other hand, AG form fits perfectly with the DC. Thus, in section 3.3, we will show that for some complex deforestation, for instance of the reverse list function composed twice, the DC directly eliminates all useless constructions, whereas this is not necessarily so with fusion methods.

3. Like for hylomorphisms, it is possible to define a transformation from a functional program into an equivalent AG. Formalizing this transformation, we have noticed strong similarities with the transformation deriving hylomorphisms from functional programs. Our algorithm transforms higher order λ-terms into recursive schemes using η-conversion. Then, we recognize recursive calls on abstract types and translate them into AG specifications. This is a very similar algorithm to that used in HYLO transformation, finding the

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

Figure 8: The *reverse* fold function

$$\begin{array}{ll}
\multicolumn{2}{l}{reverse\ (root : list) \rightarrow s_{\uparrow} : list} \\
\multicolumn{2}{l}{\texttt{Nil ->}} \\
\quad f_{Nil, s_{\uparrow Nil}} & : \ s_{\uparrow Nil} = (\lambda().Nil)\ () \\
\multicolumn{2}{l}{\texttt{Cons -> a list}} \\
\quad f_{Cons, s_{\uparrow Cons}} & : \ s_{\uparrow Cons} = (\lambda(a,r).append\ (r, Cons(a, Nil)))\ (a, s_{\uparrow list})
\end{array}$$

Figure 9: The AG corresponding to the *reverse* fold function

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

Figure 10: Second order *reverse* fold function

$$\begin{array}{ll}
\multicolumn{2}{l}{reverse\ (root : list, h_{\downarrow} : list) \rightarrow s_{\uparrow} : list} \\
\multicolumn{2}{l}{\texttt{Nil ->}} \\
\quad f_{Nil, s_{\uparrow Nil}} & : \ s_{\uparrow Nil} = Id\ (h_{\downarrow Nil}) \\
\multicolumn{2}{l}{\texttt{Cons -> a list}} \\
\quad f_{Cons, s_{\uparrow Cons}} & : \ s_{\uparrow Cons} = Id\ (s_{\uparrow list}) \\
\quad f_{Cons, h_{\downarrow list}} & : \ h_{\downarrow list} = (\lambda(a,h).Cons(a,h))\ (a, h_{\downarrow Cons})
\end{array}$$

Figure 11: The natural AG for *reverse*

underlying functor of recursive schemes. In the general case of functional programs, this transformation uses the notion of Dynamic AGs [PRJD96], as in the transformation from denotational semantics into AGs [Gan80, Lei96]

3.3 Higher Order Functions and Inherited Attributes

In this section, we deal with functional programs which require some additional transformations in order to be deforested in the fold formalism. Our running example will be the reverse list function. We will present the fold and AG approaches, and show how hylomorphisms treat these programs.

The *reverse* fold (Figure 8) is not *potentially normalizable* [SF93], because the *append* function (Figure 3) works on the accumulative result variable r which is under construction in the outer fold. For a similar reason, the DC cannot be applied over the corresponding AG (Figure 9).

To solve this problem, Sheard and Fegaras [SF93, LS95] transform this fold into a second order[2] one and introduce the *Second Order Fold Promotion Theorem* [SF93]. This transformation requires particular conditions concerning the underlying type[3], but leads to a second order fold (Figure 10) which is amenable to normalization. Furthermore, with the properties due to these type restrictions, the deforestation of *reverse* $\circ$ *reverse* leads to a *copy* function: an ideal deforestation in this case.

[2] In a second order fold, accumulative result variables can be functions.

[3] Notions of *zero-constructors* and *zero-replacement functions*: the reader will find details in [SF93] and a discussion of these restrictions in [DPRJ96]

let $(ss, hh) = (reverse \circ reverse)\ (x, Nil, hh)$
in ss
where the profile of $(reverse \circ reverse)$ 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 ->
```
$$\begin{aligned}
f_{Nil,hh_{\uparrow_{Nil}}} &: hh_{\uparrow_{Nil}} = Id\ (sh_{\downarrow_{Nil}}) \\
f_{Nil,ss_{\uparrow_{Nil}}} &: ss_{\uparrow_{Nil}} = Id\ (hs_{\downarrow_{Nil}})
\end{aligned}$$
```
Cons -> a list
```
$$\begin{aligned}
f_{Cons,sh_{\downarrow_{list}}} &: sh_{\downarrow_{list}} = Id\ (sh_{\downarrow_{Cons}}) \\
f_{Cons,hh_{\uparrow_{Cons}}} &: hh_{\uparrow_{Cons}} = (\lambda(a, hh).Cons(a, hh))\ (a, hh_{\uparrow_{list}}) \\
f_{Cons,hs_{\downarrow_{list}}} &: hs_{\downarrow_{list}} = Id\ (hs_{\downarrow_{Cons}}) \\
f_{Cons,ss_{\uparrow_{Cons}}} &: ss_{\uparrow_{Cons}} = Id\ (ss_{\uparrow_{list}})
\end{aligned}$$

Figure 12: The AG resulting from the DC of *reverse* with itself

Since AGs naturally use inherited attributes, the favorite AG form for *reverse* is not the AG in Figure 9 but rather, the one presented in Figure 11. Fortunately, the DC can then be directly applied on this natural AG (on its composition with itself). Notice here that our FP-to-AG transformation applied to the second order fold (Figure 10) leads to this AG. Moreover, as another relation between second order (fold) and inherited (AG) approaches, let us just consider Knuth's transformation [CM79], which translates any AG (with possible inherited attributes) into a purely-synthesized one. From the natural *reverse* AG with inherited attributes, this yields a purely-synthesized AG with higher order semantic rules, which corresponds to the second order fold program.

In Figure 12, we only present the AG resulting from the DC application on the example *reverse(reverse(x))*, without details on how it is obtained. Nevertheless, with inherited attributes, the DC basic idea remains the projection of semantic rules. 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 completely 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 understand it without difficulty.

The basic DC transformation leaves many copy rules between attributes (Id in Figure 12), 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 in Figure 12 yields the *copy* AG, which is equivalent to the result obtained by the NA.

In spite of the good result of the NA on this example, the particular treatment needed for higher order fold functions has lead deforestation researchers to find a more general approach. More particularly, the HYLO approach, which is more general, can deal equally well with first or higher order functions. However, encapsulation of computations in higher order functions could hide some possible "deforestable" constructions. Figure 13 presents the HYLO definition for *reverse* with *append* function, corresponding with those presented in the fold formalism

Considering this definition for reverse:

$$\begin{array}{lcl} reverse & : & ListA \rightarrow ListA \\ reverse & = & [\![Nil \triangledown \lambda(a,vs).append\ (vs, Cons(a,Nil)), Id, out_L]\!]_{L,L} \\ & = & [\![\tau in_L, Id, out_L]\!]_{L,L} \end{array}$$

where

$$\begin{array}{ll} & \tau = \lambda N \triangledown C\ .(N \triangledown (\lambda(x,r).[\![C(x,N) \triangledown C, Id, out_L]\!]_{L,L}\ r)) \\ \text{and} & F = \ !1 + !A \times I \end{array}$$

HYLO (Acid Rain Theorem) leads to:

$$reverse \circ reverse = [\![\tau(\tau in_L \circ Id), Id, out_L]\!]_{L,L}$$

Figure 13: Hylo for *reverse* with *append*

Considering this definition for reverse:

$$\begin{array}{lcl} reverse & : & ListA \rightarrow ListA \\ reverse & = & \lambda xs.rev\ xs\ Nil \\ rev & : & ListA \times ListA \rightarrow ListA \\ rev & = & [\![\phi, \eta, \psi]\!]_{F,F} = [\![Id \triangledown Id, Id, \psi]\!]_{F,F} \end{array}$$

where

$$\begin{array}{lll} & \psi = \lambda(xs,ys).\text{case } (xs,ys) \text{ of} & \\ & \quad \text{(Nil,bs)} & \rightarrow (1,(bs)) \\ & \quad \text{(Cons(a,as),bs)} & \rightarrow (2,(as, Cons(a,bs))) \\ \text{and} & F = \ !ListA + I & \end{array}$$

HYLO (Hylo-Fusion) leads to:

$$reverse \circ reverse\ xs = \lambda xs.[\![reverse \triangledown Id, Id, \psi]\!]_{F,F}\ xs\ Nil$$

Figure 14: Hylo for *reverse* with accumulating parameter

(Figure 8). Figure 14 presents another definition for *reverse*, with an accumulating parameter, corresponding with the “natural” AG. In both cases, composing *reverse* twice leads to a hylomorphism which is not a simple *copy*, unlike the DC of equivalent AG.

4 Conclusion

In the program transformation domain, there has been a recent emergence and growing interest in the structure-directed style of functional programming. Since the structure-directed paradigm is the fundamental basis of AGs, we compared functional methods and approachs related to this style with those of AGs.

We have shown that generic control operators such as fold are equivalent in expressiveness to a restricted class of attribute grammars (purely-synthesized). The NA, which performs fusion of functions, has the same effect for the first order fold, as the DC of the corresponding AG. Actually, these methods are slightly different, because we can view the DC as a more symbolic transformation than the NA, which is more akin to generalized partial evaluation. But the most important point is that they have the same local-transformation property (at type constructors level), which is possible since the type structure is explicit in programs. Moreover, we have found many similarities between extensions of such a control operator, like hylomorphisms, and the AG approach.

From the beginning, the DC has been able to handle inherited attributes, which allow more complicated programs to be expressed without higher order use. On the other hand, the NA can be applied to the same complex programs, possibly after a transformation to higher order. Since each of these approaches (higher order folds and inherited attributes) is specific to its own domain (functional programming and attribute grammars), their comparison is more difficult. In spite of obvious similarities and minor differences, this subject requires a more extensive study.

Our main motivation for this work lies in the DC important role for structure-directed genericity in AGs [DPRJ97]. Indeed, the DC is the basic tool which enables instantiation of an AG (algorithm) on a new structure (via a specification of the structure coupling by another AG). The results presented in this paper tend to prove that such a genericity is directly reusable in functional programming.

This paper reminds us —yet again— that common goal for different communities, with different approaches, may lead to fruitful cross-fertilization.

References

[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.

[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).

[DPRJ96] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2957.ps.gz`.

[DPRJ97] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

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

[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.

[FJMM91] M. M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammars to catamorphisms. *The Squiggolist*, 2(1):20–26, 1991.

[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 Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[Fok95] Maarten M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1995.

[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 (PEPM'94)*, pages 21–32, Orlando, Florida, June 1994.

[Gan80] Harald Ganzinger. Transforming denotational semantics into practical attribute grammars. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, New York–Heidelberg–Berlin, 1980.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984. Published as *ACM SIGPLAN Notices*, 19(6).

[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 Functional Programming and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeishi. Deriving structural hylomorphisms from recursive definitions. In *Proc.of the International Conference on Functional Programming (ICFP'96)*, pages 73–82, Philadelphia, May 1996. 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.

[JP92] Martin Jourdan and Didier Parigot. Application development with the FNC-2 attribute grammar system. In Dieter Hammer, editor, *Compiler Compilers '90*, Lecture Notes in Computer Science, pages 85–94. Springer-Verlag, New York–Heidelberg–Berlin, October 1990 January 1992. Schwerin.

[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*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[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).

[Lei96] Stéphane Leibovitsch. Relations entre la sémantique dénotationnelle et les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1996.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 314–323, La Jolla, CA, USA, 1995. ACM Press.

[MFP91] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144, Cambridge, September 1991. Springer-Verlag.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *In Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997.

[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, Etienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. rapport de recherche 2662, INRIA, October 1995. `ftp://ftp.inria.fr/INRIA/publication/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*, pages 263–279, Val-Morin, Québec, January 1996. ftp://ftp.inria.fr/INRIA/Projects/oscar/FNC-2/publications/jfla96.ps.gz.

[PRJD96] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 122–136, Aachen, September 1996. Springer-Verlag.

[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.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 233–242, Copenhagen, Denmark, June 1993. 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.

[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. Springer-Verlag.