

Symbolic Composition

Loïc Correnson, Etienne Duris, Didier Parigot, Gilles Roussel

N° 3348

Janvier 1998

_____ THÈME 2 _____



*R*apport
de recherche



Symbolic Composition

Loïc Correnson, Etienne Duris, Didier Parigot, Gilles Roussel*

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oscar

Rapport de recherche n° 3348 — Janvier 1998 — 24 pages

Abstract: The deforestation of a functional program is a transformation which gets rid of intermediate data structures constructions that appear when two functions are composed. The descriptonal composition, initially introduced by Ganzinger and Giegerich, is a deforestation method dedicated to the composition of two attribute grammars. This article presents a new functional deforestation technique, called symbolic composition, based on the descriptonal composition mechanism, but extending it. An automatic translation from a functional program into an equivalent attribute grammar allows symbolic composition to be applied, and then the result can be translated back into a functional program. This yields a source to source functional program transformation. The resulting deforestation method provides a better deforestation than other existing functional techniques. Symbolic composition, that uses the declarative and descriptonal features of attribute grammars is intrinsically more powerful than categorical-flavored transformations, whose recursion schemes are set by functors. These results tend to show that attribute grammars are a simple intermediate representation, particularly well-suited for program transformations.

Key-words: Deforestation, attribute grammars, functional programming, program transformation, partial evaluation.

(Résumé : tsvp)

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

Composition symbolique

Résumé : La déforestation d'un programme fonctionnel est une transformation qui vise à éliminer les constructions de structures de données intermédiaires pouvant apparaître lors de la composition de deux fonctions. La composition descriptionnelle, initialement introduite par Ganzinger et Giegerich, est une méthode de déforestation dédiée à la composition de deux grammaires attribuées. Ce rapport présente une nouvelle technique de déforestation, appelée composition symbolique, qui est une extension du mécanisme de la composition descriptionnelle. Grâce à une traduction automatique d'un programme fonctionnel en une grammaire attribuée équivalente, la composition symbolique peut être appliquée, et son résultat peut être retranscrit en un programme fonctionnel. Ceci fournit donc une transformation source à source qui peut être comparée aux autres techniques de déforestation connues. La composition symbolique, qui exploite les caractéristiques déclaratives et descriptionnelles des grammaires attribuées, est intrinsèquement plus puissante que les diverses transformations basées sur la théorie des catégories, dont les schémas de récursion sont fixés par des foncteurs. Ces résultats tendent à montrer que les grammaires attribuées sont une représentation intermédiaire simple et particulièrement adaptée aux transformations de programmes.

Mots-clé : Déforestation, grammaires attribuées, programmation fonctionnelle, transformation de programme, évaluation partielle.

1 Introduction

Intermediate data-structures are both the basis and the bane of modular programming. If they allow functions to be composed, they also have a harmful cost from efficiency point of view (allocation and deallocation). To get the best of both worlds, *deforestation* transformations were introduced. These transformations fuse two pieces of a program into another one, where intermediate data-structure constructions have been eliminated. The first approach dealing with such transformations is Wadler's [35]. It is based on the "fold and unfold" transformation [2].

There is another interesting approach based on algebraic notions and often called *deforestation in calculational form* [15, 33, 22, 34, 16]. The idea of this approach is to capture both the function and the data-type patterns of recursion [25]. The goal is to drive deforestation transformations with respect to these recursion schemes.

In attribute grammars area, *descriptive composition* [12, 14, 10, 1, 31] is a well known and powerful deforestation method which eliminates intermediate data-structure constructions in compositions. Attribute grammars [21, 27] are declarative and structure-directed specifications. They specify on each data-type pattern *what* is to be computed instead of *how* it is computed.

In [8, 9, 7, 6], we have been studying similarities and differences between descriptive composition and a large subset of deforestation methods in calculational form [15, 33, 16]. For particular¹ first-order functions, we have shown that both techniques lead to equivalent results in their respective domain. But we were also convinced that, at least for a class of programs, descriptive composition could be the basis of a more powerful tool than category-flavored deforestation methods.

As a striking example, let us consider the two following functions: *rev* reverses a list, and for a given binary tree, *flat* builds the list of its leaves from left to right (See Figure 1). In both functions, parameter *l* is initialized with *nil*:

<pre> let rev x l = case x with cons head tail → rev tail (cons head l) nil → l </pre>	<pre> let flat t l = case t with node left right → flat left (flat right l) leaf n → cons n l </pre>
--	--

The classical functional composition of these two functions leads to

$$\text{let revflat } t = \text{rev (flat } t \text{ nil) nil}$$

¹For functions that have only their pattern-matched argument as parameter.

where the list built by *flat* is the intermediate data structure consumed by *rev* (See Figure 1). Translating *rev* and *flat* into attribute grammars, our deforestation method produces a new attribute grammar that directly builds the reversed list. No longer intermediate list is constructed as presented in Figure 1.

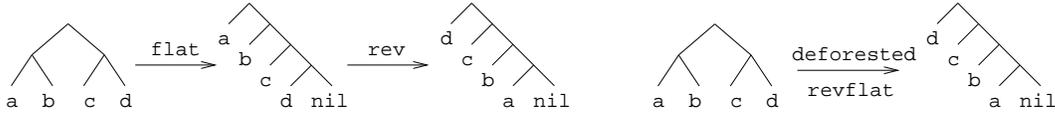


Figure 1: Example of *rev* and *flat* composition, before and after deforestation

Furthermore, it is possible to apply a *copy rule elimination*² to this deforested attribute grammar, corresponding to the following *revflat* function definition:

```
let revflat t l = case t with
  node left right → revflat right (revflat left l)
  leaf n → cons n l
```

This function directly constructs the list of leaves from right to left. The intermediate list built by the initial function *flat* is no longer constructed. As far as we know, such a deforestation cannot be achieved by any functional deforestation method.

The main contribution of this paper is to promote a transformation, based on the descriptive composition mechanism, to an efficient functional programming deforestation method. To do so, our study involves two main issues. At first, the definition of a translation, called FP-to-AG, from a functional program into its attribute grammar notation, with a new symbolic evaluation for attribute grammars; next, a projection transformation — close to the descriptive composition — combined with the symbolic evaluation, that defines a new program transformation called *symbolic composition*.

With FP-to-AG on the one hand and the reciprocal translation based on well-known technique (for instance [18]) on the other hand, the symbolic composition can be fully applied in the functional framework: it transforms a functional program into another one. This allows us to characterize a class of functional programs for which symbolic composition performs more deforestation than other functional methods.

²This optimization implies some consequences concerning the evaluation order and must be applied only in safe cases, discussed in section 3.4.

This paper is organized as follows. Section 2 defines the translation from a functional program into its attribute grammar form. Section 3 presents several transformations allowing symbolic composition to be efficiently applied on attribute grammars generated by the previous translation. Finally, section 4 deals with related work and improvements for both functional deforestation methods and attribute grammars transformations.

2 The FP-to-AG Translation

The intuitive idea for the FP-to-AG translation from a functional program into its attribute grammar notation³ is the following. Each functional term associated with a pattern has to be dismantled into a set of oriented equations, called *semantic rules*. Parameters in functional programs become explicit *attributes* attached to pattern variables, called *attribute occurrences*, that are defined by the semantic rules. Then, explicit recursive calls become implicit on the underlying data structure and semantic rules make the data-flow explicit. The FP-to-AG translation is decomposed in two steps: the *preliminary transformation* and the *profile symbolic evaluation*.

2.1 Languages and Notations

To present the basic steps of the FP-to-AG translation in a simple and clear way, we deliberately restrict ourselves to a sub-class of first order functional programs⁴, presented in Figure 2.

<i>prog</i>	$::=$	$\{def\}^*$
<i>def</i>	$::=$	$\text{let } f \bar{x} = exp$
		$\text{let } f \bar{x} = \text{case } x_k \text{ with } \{pat \rightarrow exp\}^+$
<i>pat</i>	$::=$	$c \bar{x}$
<i>exp</i>	$::=$	<i>Constant</i>
		$x \in Variables$
		$f \overline{exp}$

Figure 2: Functional language

³This notation is not the classical one, but is in a minimal form for explanatory purposes.

⁴To simplify the presentation, λ -abstractions are not allowed. By lack of space, explanation dealing with higher order functions cannot be presented in this article.

Notice that nested pattern-matching are not allowed, but it is easy to split them in several separated functions. Moreover, *if-then-else* can be taken into account with *Dynamic Attribute Grammars* [30]. We will not develop these points in this article, but other deforestation formalisms are dealing with similar classes of programs (cf. HYLO system [26]). The *rev* and *flat* programs given in introduction illustrate our functional language syntax (Figure 2).

$\begin{aligned} \text{block} & ::= \text{let } f = \{f \bar{x} \rightarrow \overline{\text{semrule}}\} \{pat \rightarrow \overline{\text{semrule}}\}^* \\ \text{semrule} & ::= \text{occ} = \text{exp} \\ \text{occ} & ::= x.a \mid a \end{aligned}$
<p>Note that <i>exp</i> is as for Fig. 2 and that <i>occ</i> is added to <i>Variables</i>.</p>

Figure 3: Attribute grammar notation

To bring our attribute grammar notation, presented in Figure 3, closer to functional specifications, algebraic type definitions will be used instead of classical context free grammars [3, 12]. For example, types *list* and *tree* are defined as follows:

$$\begin{array}{lcl} \text{list } \alpha & = & \text{cons } \alpha \text{ (list } \alpha) \\ & | & \text{nil} \end{array} \qquad \begin{array}{lcl} \text{tree } \alpha & = & \text{node (tree } \alpha) \text{ (tree } \alpha) \\ & | & \text{leaf } \alpha \end{array}$$

A grammar production is represented as a data-type constructor followed by its parameter variables, that is, a pattern (for example: *cons head tail*).

Since our transformations take type-checked functional programs as input, this induces information about the generated attribute grammars. For example, a distinctive feature of attribute grammars is to characterize two sorts of attributes: the *synthesized* ones are computed bottom-up over the structure and the *inherited* ones are computed top-down. The sort and the type of an attribute are directly deduced from the type-checked input program⁵.

Furthermore, the notion of *attribute grammar profile* is introduced (in Figure 3, $f \bar{x}$ is the profile of f). It represents how to call the attribute grammar and allows result and arguments to be specified. This notion freely extends classical attribute grammars where these argument specifications were impossible⁶.

The occurrence of an attribute a on a pattern variable x is noted $x.a$. If an attribute is attached to the constructor of the current pattern, its occurrence is simply

⁵This frees our attribute grammar notation from these information.

⁶Some similar attempts exist, like in [13].

noted a ⁷. For instance, according to the attribute grammar syntax in Figure 3, the *rev* function is defined as follows:

```

let rev =                               ; Name of the attribute grammar.
                                           ; Profile of the attribute grammar:
rev x l →                               ; x is the pattern-matched argument, l the parameter.
  result = x.rev                         ; result is the only synthesized attribute of the profile.
  x.lrev = l                             ; x has 2 attributes: rev synthesized and lrev inherited.
cons head tail →                       ; Pattern matching on the cons constructor.
  rev = tail.rev                         ; Attribute occurrence rev definition (bottom-up).
  tail.lrev = cons head lrev             ; Attribute occurrence tail.lrev definition (top-down).
nil →                                    ; Pattern matching on the nil constructor.
  rev = lrev                             ; Attribute occurrence rev definition with lrev.

```

In further transformation algorithms, the following notations are used:

$\underline{\underline{def}}$:	local definition in an algorithm
\bar{x}	:	a n-tuple x_1, \dots, x_n
$x.a = exp$:	semantic rule defining the attribute occurrence $x.a$
$[x := y]$:	substitution of x by y
Σ	:	a set of semantic rules
Π	:	a pattern with its set of semantic rules
$\mathcal{C} \vdash A \Rightarrow B$:	transformation from A into B according to the context \mathcal{C}
$\mathcal{E}[e]$:	a term containing e as a sub-expression.

2.2 Preliminary Transformation

The aim of the *preliminary transformation*, presented in Figure 4, is to draw the general shape of the future attribute grammar. It introduces the attribute grammar profile with its semantic rules, and a unique semantic rule per each constructor pattern.

The attribute *result* is defined as a synthesized attribute of the attribute grammar profile and contains the result of the function (rule *Let'*). For function with **case** statement the result is computed through attributes on the pattern-matched variable (rule *Let*). Other arguments are translated into semantic rules defining

⁷ a attached to the current constructor could be view as a contraction of $this.a$, that will be used in algorithms for clarity.

$\frac{\forall i \quad \overset{exp}{\vdash} a_i \Rightarrow b_i \quad ; \quad f \text{ is a function name}}{\overset{exp}{\vdash} (f \bar{a}) \Rightarrow (f \bar{b}).result} \quad (App)$
$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{\quad} \quad (Pattern)$
$f, \{x_j\}_{j \neq k}, x_k \overset{pat}{\vdash} c \bar{y} \rightarrow e \Rightarrow c \bar{y} \rightarrow f = e'[x_k := this][x_j := this.x_j^f]_{j \neq k}$
$\frac{\forall i \quad f, \{x_j\}_{j \neq k}, x_k \overset{pat}{\vdash} p_i \rightarrow e_i \Rightarrow \Pi_i \quad \bar{\Pi} \stackrel{def}{=} \left(\begin{array}{l} f \bar{x} \rightarrow \\ result = x_k.f \\ x_k.x_j^f = x_j \quad (j \neq k) \end{array} \right) \cup \Pi_i}{\overset{let}{\vdash} \text{let } f \bar{x} = \text{case } x_k \text{ with } \bar{p} \Rightarrow \bar{e} \Rightarrow \text{let } f = \bar{\Pi}} \quad (Let)$
$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{\overset{let}{\vdash} \text{let } f \bar{x} = e \Rightarrow \text{let } f = f \bar{x} \rightarrow result = e'} \quad (Let')$
<p>$\overset{exp}{\vdash} e \Rightarrow e'$ means that the equation e is translated into the equation e'.</p> <p>$env \overset{pat}{\vdash} p \rightarrow e \Rightarrow p \rightarrow \mathcal{R}$ means that the expression associated with the pattern p is translated into the set of semantic rules \mathcal{R}, with respect to the environment env.</p> <p>$\overset{let}{\vdash} \mathcal{D} \Rightarrow \mathcal{B}$ means that the function definition \mathcal{D} is translated into the block \mathcal{B}.</p>

Figure 4: Preliminary transformation

inherited attributes attached to the pattern-matched variable (rule *Let*). Each function call $(f \bar{a})$ is translated into a dotted notation $(f \bar{b}).result$ (rule *App*). This rule distinguishes between function and type constructor calls⁸. Each expression appearing in a pattern is transformed into a semantic rule that defines the synthesized attribute computing the result (rule *App*). This induces some renaming (rule *Pattern*).

The application of the preliminary transformation to the *flat* function leads to:

```

let flat =
  flat t l →
    result = t.flat
    t.lflat = l
  node left right →
    flat = (flat left (flat right lflat).result).result
  leaf n →
    flat = cons n lflat

```

2.3 Profile Symbolic Evaluation

The result of the preliminary transformation is not yet a real attribute grammar. Each function definition in the initial program has been translated into one *block* (cf. Figure 3) that contains the profile of the function and its related patterns. But explicit recursive calls have been translated into the form $(f \bar{a}).result$. Now, these expressions have to be transformed into a set of more detailed semantic rules, breaking explicit recursions by attribute naming and attachment to pattern variables. Then, semantic rules will implicitly define the recursion “à la” attribute grammar. This transformation is achieved by the *profile symbolic evaluation*, presented in Figure 5.

Everywhere an expression $(f \bar{a}).result$ occurs, the profile symbolic evaluation projects the semantic rules of the attribute grammar profile f . The application of this transformation must be done with a depth-first application strategy. The *Check* constraint ensures that the resulting attribute grammar is well formed. Essentially, it verifies that each attribute is defined once and only once. This is generally the case since parameters in input functional programs are well-defined, but *Check* forbid some non-linear terms such that $g (f x 1) (f x 2)$. Moreover, in a first approach, terms like $(x.a).b$ are not allowed but they will be treated in section 3.2. Wherever

⁸This distinction is performed using type information from the input functional program.

$$\begin{array}{c}
\left(\begin{array}{l} f \bar{x} \rightarrow \\ \text{result} = \varphi \\ \Sigma_f \end{array} \right) \in \mathcal{P} \quad \sigma \stackrel{\text{def}}{=} [x_i := a_i] \\
\Sigma \stackrel{\text{def}}{=} \left\{ \begin{array}{l} u = \mathcal{E}[\sigma(\varphi)] \\ \sigma(\Sigma_f) \\ \Sigma_{aux} \end{array} \right. \quad \text{Check}(c, f, \Sigma) \\
\hline
\mathcal{P} \vdash \left(\begin{array}{l} c \bar{y} \rightarrow \\ u = \mathcal{E}[(f \bar{a}).\text{result}] \\ \Sigma_{aux} \end{array} \right) \Rightarrow \left(\begin{array}{l} c \bar{y} \rightarrow \\ \Sigma \end{array} \right) \\
\mathcal{P} \vdash p \rightarrow \Sigma_1 \Rightarrow p \rightarrow \Sigma_2 \quad \text{means that in the program } \mathcal{P} \text{ the set of equations } \Sigma_1 \\
\text{of a pattern } p \text{ is transformed into } \Sigma_2.
\end{array} \quad (PSE)$$

Figure 5: Profile Symbolic Evaluation

$\text{Check}(c, f, \Sigma)$ is not verified, the expression $(f \bar{a}).\text{result}$ is simply rewritten in the function call $(f a)$.

In the previous *flat* example, the semantic rule associated to pattern *node left right* is

$$\text{flat} = (\text{flat left } (\text{flat right } l^{\text{flat}}).\text{result}).\text{result}$$

The application⁹ of (PSE) rule on this semantic rule is:

$$\begin{array}{c}
\left(\begin{array}{l} \text{flat } t \ l \ \rightarrow \\ \{ \text{result} = t.\text{flat} ; t.l^{\text{flat}} = l \} \end{array} \right) \in \mathcal{P} \\
\sigma \stackrel{\text{def}}{=} [t := \text{right}][l := l^{\text{flat}}] \\
\Sigma \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \text{flat} = (\text{flat left } \text{right}.\text{flat}).\text{result} \\ \text{right}.l^{\text{flat}} = l^{\text{flat}} \end{array} \right. \\
\text{Check}(\text{node}, \text{flat}, \Sigma) \\
\hline
\text{node left right } \rightarrow \\
\text{flat} \vdash \text{flat} = (\text{flat left } (\text{flat right } l^{\text{flat}}).\underline{\text{result}}).\text{result} \\
\Rightarrow \text{node left right } \rightarrow \Sigma
\end{array} \quad (PSE)$$

⁹Underlined *terms* show where the rule is being applied.

Finally, complete application of the profile symbolic evaluation leads for the function *flat* to the well-formed attribute grammar given below. The successive application of preliminary transformation and profile symbolic evaluation to an input functional program leads to a real attribute grammar. This is the FP-to-AG translation.

```

let flat =
  flat t l →
    result = t.flat
    t.lflat = l
  node left right →
    flat = left.flat
    left.lflat = right.flat
    right.lflat = lflat
  leaf n →
    flat = cons n lflat

```

3 Symbolic Composition

Since attribute grammars could be obtained from functional programs, it is possible to apply attribute grammars deforestation methods, like the descriptive composition. To be able to present our symbolic composition, we first present a natural extension of profile symbolic evaluation that is useful in the application of the symbolic composition.

It is important to note here that even if the final results of symbolic composition are attribute grammars, the objects that will be manipulated by intermediate transformations are more *blocks* of attribute grammars rather than complete attribute grammars. Furthermore, the expressions of the form $(x.a).b$ previously avoided (cf. *Check* predicate in section 2.3), will be temporarily authorized for the symbolic composition process.

3.1 Symbolic Evaluation

Profile symbolic evaluation can be generalized into a new *symbolic evaluation* that performs both profile symbolic evaluation and partial evaluation on constant terms. The idea of this symbolic evaluation is to project recursively semantic rules on finite terms and to eliminate intermediate attributes which are defined and used in the produced set of semantic rules. Figure 6 describes this transformation.

$$\boxed{
\begin{array}{c}
\left(\begin{array}{l} f \bar{x} \rightarrow \\ w = \varphi \\ \Sigma_f \end{array} \right) \in \mathcal{P} \quad \sigma \stackrel{def}{=} [x_i := a_i][h := \varphi_h]_h \\
\quad \quad \quad \Sigma \stackrel{def}{=} \left\{ \begin{array}{l} u = \mathcal{E}[\sigma(\varphi)] \\ \sigma(\Sigma_f) \\ \Sigma_{aux} \end{array} \right. \quad Check(c, f, \Sigma) \\
\hline
\mathcal{P} \vdash \left(\begin{array}{l} c \bar{y} \rightarrow \\ u = \mathcal{E}[(f \bar{a}).w] \\ (f \bar{a}).h = \varphi_h \\ \Sigma_{aux} \end{array} \right) \Rightarrow \left(\begin{array}{l} c \bar{y} \rightarrow \\ \Sigma \end{array} \right)
\end{array}
\quad (SE)$$

Figure 6: Symbolic Evaluation

To illustrate the use of symbolic evaluation as partial evaluation, consider the term $(rev (cons a (cons b nil)) nil)$. The profile symbolic evaluation (Figure 5) applied on this term yields the two following semantic rules:

$$\begin{aligned}
result &= (cons a (cons b nil)).rev \\
(cons a (cons b nil)).l^{rev} &= nil
\end{aligned}$$

Then, the symbolic evaluation (Figure 6) could be applied on these terms. The first step of this application is presented below:

$$\begin{array}{c}
\left(\begin{array}{l} cons \ head \ tail \rightarrow \\ rev = tail.rev \\ tail.l^{rev} = cons \ head \ l^{rev} \end{array} \right) \in \mathcal{P} \\
\sigma = [head := a][tail := cons \ b \ nil][l^{rev} = nil] \\
\Sigma = \left\{ \begin{array}{l} result = (cons \ b \ nil).rev \\ (cons \ b \ nil).l^{rev} = cons \ a \ nil \end{array} \right. \\
Check(c, cons, \Sigma) \\
\hline
\mathcal{P} \vdash \quad c \bar{y} \rightarrow \left\{ \begin{array}{l} result = (cons \ a \ (cons \ b \ nil)).rev \\ (cons \ a \ (cons \ b \ nil)).l^{rev} = nil \end{array} \right\} \Rightarrow c \bar{y} \rightarrow \Sigma
\end{array}
\quad (SE)$$

Two other steps of this transformation lead to the term

$$result = (cons \ b \ (cons \ a \ nil))$$

So, symbolic evaluation performs **partial evaluation** of finite terms.

3.2 Composition

Getting back to our first example, let us consider the definition of the function *revflat* which flattens a tree and then reverses the obtained list. In this composition the result of *flat* is a deforestationable intermediate list, since it is consumed by *rev*.

$$\text{let } revflat\ t = (rev\ (flat\ t\ nil)\ nil)$$

Intuitively, in the context of our attribute grammar notation, the composition of *rev* and *flat* involves the two following sets of attributes:

$$Att_{flat} = \{flat, l^{flat}\} \quad \text{and} \quad Att_{rev} = \{rev, l^{rev}\}$$

More generally, consider an attribute grammar, say \mathcal{F} (e.g., *flat*), producing an intermediate data structure to be consumed by another attribute grammar, say \mathcal{G} (e.g., *rev*). Two sets of attributes are involved in this composition. The first one, $Att_{\mathcal{F}}$, contains all the attributes used to construct the intermediate data-structure. The second one, $Att_{\mathcal{G}}$, contains the attributes of \mathcal{G} .

As in the descriptonal composition, the idea of the symbolic composition is to project the attributes of $Att_{\mathcal{G}}$ (e.g., Att_{rev}) everywhere an attribute of $Att_{\mathcal{F}}$ (e.g., Att_{flat}) is defined. This global operation brings the equations that specify a computation over the intermediate data-structure on its construction. The basic step of this projection is presented in Figure 7. Then, the application of the symbolic evaluation (Figure 6) will eliminate the useless constructors.

$\frac{\begin{array}{l} a \in Att_{\mathcal{F}} \\ \bar{s} = Att_S_{\mathcal{G}} \\ \bar{h} = Att_H_{\mathcal{G}} \end{array}}{Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash x.a = e \Rightarrow \begin{cases} (x.a).s = (e).s & \forall s \in \bar{s} \\ (e).h = (x.a).h & \forall h \in \bar{h} \end{cases}} \quad (Proj)$
<p>$Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash eq \Rightarrow \Sigma$ means that, while considering $\mathcal{G} \circ \mathcal{F}$, the equation eq is transformed into the set of equations Σ.</p>
<p style="margin-left: 40px;">$Att_S_{\mathcal{G}}$ is the set of synthesized attributes of $Att_{\mathcal{G}}$.</p>
<p style="margin-left: 40px;">$Att_H_{\mathcal{G}}$ is the set of inherited attributes of $Att_{\mathcal{G}}$.</p>

Figure 7: Projection step

However, a point remains undefined: how find the application sites for the projection steps? As attended, the constraint in the *Check* predicate avoiding expressions like $(x.a).b$ to arise is temporarily relaxed. In fact, all these expressions are precisely the sites where deforestation could be performed (e.g., $(t.flat).rev$).

With this relaxed *Check* predicate, and from the *revflat* function definition, we obtained the following blocks.

This block is for the <i>revflat</i> profile	$ \begin{array}{l} \text{revflat } t \rightarrow \\ \text{result} = (t.flat).rev \\ (t.flat).l^{rev} = nil \\ \underline{t.l^{flat} = nil} \quad * \end{array} $
These blocks correspond to the <i>flat</i> attribute grammar	$ \begin{array}{l} \text{node } left \ right \rightarrow \\ \underline{flat = left.flat} \\ \underline{left.l^{flat} = right.flat} \\ \underline{right.l^{flat} = l^{flat}} \\ \text{leaf } n \rightarrow \\ \underline{flat = cons \ n \ l^{flat}} \quad * \end{array} $
These blocks correspond to the <i>rev</i> attribute grammar	$ \begin{array}{l} \text{cons } head \ tail \rightarrow \\ rev = tail.rev \\ tail.l^{rev} = cons \ head \ l^{rev} \\ nil \rightarrow \\ rev = l^{rev} \end{array} $

In the blocks building the intermediate data structure, the application sites for the projection step are underlined, and a $*$ highlights the actual constructions to be deforested, that is, where stand *cons* and *nil* that build the intermediate list.

In order to illustrate the (*Proj*) step, its application on the pattern *leaf n* is given below.

$$\frac{
 \begin{array}{l}
 flat \in Att_{flat} \\
 \bar{s} = Att_S_{rev} = \{rev\} \\
 \bar{h} = Att_H_{rev} = \{l^{rev}\}
 \end{array}
 }{
 Att_{rev}, Att_{flat} \vdash flat = cons \ n \ l^{flat}
 } \quad (Proj)$$

$$\Rightarrow \left\{ \begin{array}{l}
 flat.rev = (cons \ n \ l^{flat}).rev \\
 (cons \ n \ l^{flat}).l^{rev} = flat.l^{rev}
 \end{array} \right.$$

Applying this projection step to all possible sites yield the following blocks:

$$\begin{array}{l}
\text{revflat } t \rightarrow \\
\quad \text{result} = (t.\text{flat}).\text{rev} \\
\quad (t.\text{flat}).l^{\text{rev}} = \text{nil} \\
\quad (t.l^{\text{flat}}).\text{rev} = (\text{nil}).\text{rev} \\
\quad (\text{nil}).l^{\text{rev}} = (t.l^{\text{flat}}).l^{\text{rev}} \quad \left. \vphantom{\begin{array}{l} (t.l^{\text{flat}}).\text{rev} \\ (\text{nil}).l^{\text{rev}} \end{array}} \right\} \text{ site for SE application} \\
\text{node left right} \rightarrow \\
\quad \text{flat.rev} = (\text{left.flat}).\text{rev} \\
\quad (\text{left.flat}).l^{\text{rev}} = \text{flat}.l^{\text{rev}} \\
\quad (\text{left.l}^{\text{flat}}).\text{rev} = (\text{right.flat}).\text{rev} \\
\quad (\text{right.flat}).l^{\text{rev}} = (\text{left.l}^{\text{flat}}).l^{\text{rev}} \\
\quad (\text{right.l}^{\text{flat}}).\text{rev} = (l^{\text{flat}}).\text{rev} \\
\quad (l^{\text{flat}}).l^{\text{rev}} = (\text{right.l}^{\text{flat}}).l^{\text{rev}} \\
\text{leaf } n \rightarrow \\
\quad \left. \begin{array}{l} \text{flat.rev} = (\text{cons } n \text{ } l^{\text{flat}}).\text{rev} \\ (\text{cons } n \text{ } l^{\text{flat}}).l^{\text{rev}} = \text{flat}.l^{\text{rev}} \end{array} \right\} \text{ site for SE application}
\end{array}$$

Now, symbolic evaluation (Figure 6) could be applied on annotated sites above, performing the actual deforestation. New attributes are created by renaming attributes $a.b$ into a_b (when $a \in \text{Att}_{\mathcal{F}}$ and $b \in \text{Att}_{\mathcal{G}}$). More precisely, $(x.a).b$ is transformed into $x.a_b$.

Then, the basic constituents of the symbolic composition are defined:

$$\boxed{\text{Symbolic Composition} = \text{renaming} \circ (\text{SE}) \circ (\text{Proj})}$$

Thus, for the *revflat* function, the symbolic composition yields the deforested attribute grammar that is presented in Figure 8, together with its equivalent function definition, corresponding to a functional evaluator generated for this attribute grammar [18, 29].

Four attributes have been generated. The final list is constructed with $l^{\text{flat}}_l^{\text{rev}}$ and $\text{flat}_l^{\text{rev}}$; this construction corresponds to the function *revflat1*. Then it is propagated backward, with $\text{flat}_r^{\text{rev}}$ and $\text{flat}_l^{\text{rev}}$, before being assigned to *result*; this corresponds to the function *revflat2*.

We will present in section 3.4 a way to eliminate most of these propagations, but henceforth, **no longer intermediate list is constructed**.

```

let revflat =
  revflat t →
    result = t.flat_rev
    t.flat_lrev = nil
    t.lflat_rev = t.lflat_lrev
  node left right →
    flat_rev = left.flat_rev
    left.flat_lrev = flat_lrev
    left.lflat_rev = right.flat_rev
    right.flat_lrev = left.lflat_lrev
    right.lflat_rev = lflat_rev
    lflat_lrev = right.lflat_lrev
  leaf n →
    flat_rev = lflat_rev
    lflat_lrev = cons n flat_lrev

let revflat t l = revflat2 t (revflat1 t l)
  where
    let revflat1 t l = case t with
      node left right →
        revflat1 right (revflat1 left l)
      leaf n →
        cons n l
    and
    let revflat2 t l = case t with
      node left right →
        revflat2 left (revflat2 right l)
      leaf n →
        l

```

Figure 8: The deforested *revflat* attribute grammar and its equivalent function definition

3.3 Applying Symbolic Composition

We have presented the symbolic composition on simple cases. For more complex programs, the result of this transformation could possibly be a ill-formed attribute grammar, because of the following remarks, essentially corresponding to the constraints introduced by Giegerich and Ganzinger about descriptonal composition for classical attribute grammars.

The projections in symbolic composition must be performed only on terms that *participate* to the construction of the intermediate data structure. This is the problem of determining the set $Att_{\mathcal{F}}$, which corresponds to the Ganzinger and Giegerich's separation between syntactic and semantic domains. This induces that the complete construction of the intermediate data structure ought to be available, and must not be hidden. Moreover, in the resulting attribute grammar, each attribute occurrence must be defined only once. Such problem could arise with some non-linear terms.

The Ganzinger and Giegerich's constraints could be used in a first approach to resolve these problems. Nevertheless, our special context of type-checked functional

programs permits to reformulate the resolution of these problems in terms of a particular static analysis. From our point of view, the information required by the composition mechanism must be determined separately. This independence had allowed us to present our symbolic composition, even if the problem of static analysis in functional context constitutes an interesting study, not tackled in this paper.

3.4 Copy Rules Elimination

Attribute grammars, and particularly those generated by symbolic composition, may contain many unnecessary attribute copy rules. They are only carrying values or constants around the input structure. However, a simple static global analysis on the attribute grammar can eliminate them in most cases [31].

In the case of the *revflat* example, the deforestation is due to the fact that the result is already evaluated in the attribute $l^{flat_l^{rev}}$ and then passed along the tree via attribute l^{flat_rev} to *flat_rev*. At this point, the symbolic composition is completely semantic preserving, even in the case of partially defined tree, with one infinite branch, for instance. The deforestation is complete in the sense of intermediate data structure elimination. Nevertheless, in safe cases, i.e. totally defined trees without infinite or undefined branch, the last traversal can be removed. In each node, the equality between attributes *flat_rev*, l^{flat_rev} and $l^{flat_l^{rev}}$ can be proven by structural induction over a tree. So, for this example, the copy rules elimination [31] leads to the attribute grammar below. This last version only constructs the list of the leaves from right to left without useless traversal around the tree. Finally, generating a functional evaluator [18, 29] for this attribute grammar will lead to the deforested function *revflat* expected in introduction.

```

let revflat =
  revflat t →
    result = t.lflat_lrev
    t.flat_lrev = nil
  node left right →
    lflat_lrev = right.lflat_lrev
    left.flat_lrev = flat_lrev
    right.flat_lrev = left.lflat_lrev
  leaf n →
    lflat_lrev = cons n flat_lrev

```

4 Related Work and Results Interpretation

We have already shown in [8, 9] that for simple programs, and particularly S^1 attribute grammars (that can be computed with only one synthesized attribute), descriptional composition and functional deforestation led to equivalent results.

In spite of the restrictions associated with our FP-to-AG translation, this paper shows that programs like $(rev (flat t nil) nil)$ or $rev (rev x nil) nil$ are deforested by symbolic composition while they are not by functional deforestations in calculational form. So we can formulate our main result as :

$$\boxed{SC \circ FP\text{-to-AG} > FP\text{-to-AG} \circ \text{Functional-Deforestation}}$$

In section 4.1, we try to point out some reasons to explain why symbolic composition seems intrinsically stronger than the various category-flavored methods.

In fact, most structure-directed methods in functional programming are based on categorical notions such as functors, catamorphisms and hylomorphisms. These methods are supported by fundamental laws like *Promotion Theorem* [25], and use generic control operators to capture both the function and the type definition patterns of recursion. First, *shortcut deforestation* [15] with `foldr/buildr` elimination rule made this possible for the type list. Then, the *Normalization Algorithm* [33, 11, 22] generalized it to work on any type, thanks to an automatic generation of functors from algebraic type definitions. But these functors were too much isomorphic to the types. To relax this constraint, hylomorphisms in triplet form [34] were introduced but they still deal with functors. Systems like ADL and HYLO [26, 20] are based on this formalism and deforest some complex functional programs using an automatic process.

4.1 Deforestation improvement

In spite of all these successive refinements and generalizations, one class of programs remains undeforested (e.g., $rev \circ rev$ and $rev \circ flat$ belong to this class). From our point of view, the following informal remarks could help to characterize this class.

Functional methods always use functors to drive transformations and computations, while symbolic composition and attribute grammars do not. The example of rev is meaningful:

```
let rev x l = case x with
  cons head tail → rev tail (cons head l)
  [...]
```

Let F_{rev} be the functor that drives the recursion of this function. F_{rev} does not follow the construction of the resulting list of reverse. But following the recursive calls of *rev*, it hides the construction of the result. More precisely, the constructor *cons* is hidden in the second parameter of the *rev* call. Because of this, no further deforestation can reach it.

In attribute grammars, symbolic composition catches each constructor of the result, directly from the specification. It does not need to do this using any particular abstract intermediate notion, such as functors. The reason is that all the result constructions, even if they do not follow the functor of the recursion, remain visible in an attribute grammar specification. We believe this is why symbolic composition is able to perform more deforestation. See for example the section 3.2 where the previous *cons* is deforested. To deforest such programs, functional approaches should use a functor that describes completely the result construction scheme, particularly taking into account accumulative parameters in which constructions or transmissions of — parts of — the result occur.

To conclude, one of the important contribution of this paper is to show that symbolic composition is actually a *general* deforestation method. Thanks to FP-to-AG, symbolic composition is no longer restricted to attribute grammars.

4.2 Attribute Grammars improvement

For the attribute grammars domain, this work provides a more complete integration of descriptional composition in a more usable way. The initial idea introduced by Ganzinger and Giegerich was to apply descriptional composition to two separate attribute grammars. The symbolic composition allows deforestation to be performed on terms inside an attribute grammar. Moreover, all finite terms are evaluated through these transformations. In this context, partial evaluation becomes some special case of symbolic composition.

Finally, recall that the attribute grammar formalism is not only an abstract notation for writing semantic equations. It is also itself a complete programming language, well known for its power and efficiency in writing large applications such as compilers. We have much experience in this area with the FNC-2 system [19]. Furthermore, to perform deforestation — even in complex situations — no extension of the initial and simple formalism is needed.

Thus, we believe that attribute grammars could be used profitably as an alternative to other classical formalisms for the deforestation purpose.

5 Conclusion

The main goal of this article is to show that it is possible to translate (interpret) attribute grammars transformation techniques in functional programs transformation terms.

More precisely, we show that thanks to the fundamental algorithm of the symbolic composition, our deforestation achieves better results than other techniques developed in the functional community. This result reinforces our conviction that the attribute grammar formalism is simple and powerful for this kind of transformation. The FP-to-AG translation (presented here in its simpler form) together with the reciprocal Johnsson's transformation should be viewed as auxiliary tools. For a practical use of this deforestation, FP-to-AG could be improved and extended, but this will not question the intrinsic power of our symbolic composition. Furthermore, this problem is not specific to our attribute grammar-based deforestation since it arises also in calculational systems, such as HYLO (see for example [26]).

Moreover, we extend the basic descriptonal composition into a more powerful one: the symbolic composition. This extension allows it on the one hand to be used as a partial evaluation and on the other hand to be more widely usable. Henceforth, it could be applied to terms with function compositions, and not only to one composition of two distinct attribute grammars (grammar couple [14]) that are isolated of all context. From the point of view of the attribute grammars community, this should stand as the main contribution of this article.

This work is a part of a more general study, that is the *genericity with attribute grammars*. The principle of this kind of genericity, whose basic tool is the symbolic composition, is to abstract a program, to be able to specialize it in several contexts. Similar approaches are being studying in different programming paradigms (*polytypic programming* [17], *adaptive programming* [28]). We just begin to compare [5] these approaches with our genericity tools [23, 24, 32, 31, 4], that have been implemented in our FNC-2 system. It appears also in this context that attribute grammars, particularly suitable for program transformations, should be viewed more as an abstract representation of a specification than as a programming language.

Acknowledgments

We are grateful to Dick Kieburtz for providing encouragement after fruitful discussions on this work. Many thanks also to Françoise Bellegarde and John Tang Boyland for their useful comments on draft versions of this paper.

References

- [1] John Boyland and Susan L. Graham. Composing tree attributions. In *21st ACM Symp. on Principles of Programming Languages*, pages 375–388, Portland, Oregon, January 1994. ACM Press.
- [2] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
- [3] 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).
- [4] Loïc Correnson. Généricité dans les grammaires attribuées. Rapport de stage d’option, École Polytechnique, 1996.
- [5] Loïc Correnson. Programmation polytypique avec les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1997.
- [6] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute grammars and functional programming deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, Paris, France, September 1997.
- [7] Etienne Duris. Functional programming and attribute grammar deforestation. In *Proc. of the International Conference on Functional Programming (ICFP’97) – Poster Session*, Amsterdam, The Netherlands, June 1997. ACM Press.
- [8] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996.
- [9] 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.
- [10] 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.

- [11] 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.
- [12] 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).
- [13] Harald Ganzinger, Robert Giegerich, and Martin Vach. MARVIN: a tool for applicative and modular compiler specifications. Forschungsbericht 220, Fachbereich Informatik, University Dortmund, July 1986.
- [14] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.
- [15] 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.
- [16] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeishi. Deriving structural hylo-morphisms from recursive definitions. In *Proc. of the International Conference on Functional Programming (ICFP'96)*, pages 73–82, Philadelphia, May 1996. ACM Press.
- [17] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *24th ACM Symp. on Principles of Programming Languages*, 1997.
- [18] 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.
- [19] Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Evaluation Methods*, volume 545 of *Lect. Notes in Comp. Sci.*, pages 485–504, New York–Heidelberg–Berlin, June 1991. Springer-Verlag. Prague.
- [20] Richard Kieburtz and Jeffrey Lewis. Algebraic design language. Technical report, Oregon Graduate Institute, 1994.

-
- [21] 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).
- [22] 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.
- [23] Carole Le Bellec. *La généralité et les grammaires attribuées*. PhD thesis, Département de Mathématiques et d’Informatique, Université d’Orléans, 1993.
- [24] 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, August 1993. Springer-Verlag.
- [25] 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.
- [26] 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.
- [27] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
- [28] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linköping, Sweden, 1996. Springer Verlag.
- [29] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. Rapport de Recherche 2662, INRIA, October 1995.
- [30] 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.
- [31] 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.
- [32] Gilles Roussel, Didier Parigot, and Martin Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzsion, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *Lect. Notes in Comp. Sci.*, pages 52–67, Edinburgh, April 1994. Springer-Verlag.
- [33] 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.
- [34] 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.
- [35] 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.



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, Irista, 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)
<http://www.inria.fr>
ISSN 0249-6399