INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

How to Deforest in Accumulative Parameters?

Loïc Correnson, Etienne Duris, Didier Parigot and Gilles Roussel

No 3608

Janvier 1999

THÈME 2

Rapport
de recherche

ISSN 0249-6399

How to Deforest in Accumulative Parameters?

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

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oscar

Rapport de recherche n° 3608 — Janvier 1999 — 20 pages

Abstract: Software engineering has to reconcile modularity with efficiency. One way to grapple with this dilemma is to automatically transform a modular-specified program into an efficient-implementable one. This is the aim of deforestation transformations which get rid of intermediate data structures constructions that appear when two functions are composed. Nevertheless, existing functional methods cannot deforest non-trivial intermediate constructions that are processed by symbolic composition. This new deforestation technique is based on the descriptional composition dedicated to attribute grammars. In this paper, we present the symbolic composition, we outline its counterpart in terms of classical deforestation methods and we sketch a way to embed it in a functional framework.

Key-words: Attribute grammars, functional programming, deforestation, program transformation.

(Résumé : tsvp)

* Etienne Duris is with Cedric-IIE, CNAM. E-mail: `duris@iie.cnam.fr`
† Gilles Roussel is with Université de Marne-la-Vallée. E-mail: `roussel@univ-mlv.fr`

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Comment déforester dans les paramètres d'accumulation ?

Résumé :
Le développement logiciel doit réconcilier la modularité et l'efficacité. Une des approches pour s'attaquer à ce dilemme est de permettre la transformation automatique de programmes spécifiés modulairement en programmes implémentables efficacement. C'est précisément le but des transformations de déforestation, qui permettent d'éliminer des constructions de structures de données intermédiaires apparaissant lors de la composition de deux fonctions. Néanmoins, les méthodes fonctionnelles existantes ne peuvent pas déforester certaines constructions intermédiaires non triviales qui sont pourtant prises en compte par la composition symbolique. Cette nouvelle technique de déforestation est fondée sur le principe de la composition descriptionnelle qui est dédiée aux grammaires attribuées. Dans ce rapport, nous présentons la composition symbolique, nous précisons ses contreparties pour les méthodes de déforestation classiques et nous esquissons une méthode permettant de l'insérer dans un schéma de programmation fonctionnelle.

Mots-clé : Grammaires attribuées, programmation fonctionnelle, déforestation, transformation de programmes.

1 Introduction

Intermediate data-structures are both the basis and the bane of modular programming [25]. 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 source-to-source transformations fuse two pieces of a program into another one, where intermediate data-structure constructions have been eliminated.

Our studies related to program transformations with the formalism of attribute grammars led us to develop a new deforestation mechanism, based on the classical descriptional composition of attribute grammars [6, 8, 21]. Since it processes particularly powerful deforestations, the aim of this paper is to present this *symbolic composition* together with a way to embed it in a functional framework. Furthermore, in order to bridge the gap between two communities, we try to outline an interpretation of our method in terms of functional deforestation techniques.

After a large motivating overview of the problem in this introduction, we detail the successive steps allowing our deforestation method to be processed on functional programs: a translation from functional programs to attribute grammars and the core of the symbolic composition. Classical techniques could then translate the resulting deforested attribute grammars into functional programs (by the way of functional evaluators [19], or lazy evaluation [12]).

1.1 Different approaches

The Wadler's deforestation algorithm [25] is based on the Burstall and Darligton's "unfold/fold" transformation [1]. Since functions are defined with respect to data type constructors, manipulations of their definitions allow constructors – of an intermediate data structure – to be exposed to functions that consume them. Then, the transformation consists in shifting constructors outside the treatments of internal functions, i.e delaying their actual constructions.

There is another interesting approach based on algebraic notions and often called *deforestation in calculational form* [9, 23, 14, 24, 10]. The idea of this approach is to capture both the function and the data-type patterns of recursion [16]. The goal is to drive deforestation transformations with respect to these recursion schemes and to classical laws of the category theory.

All these methods are able to deforest function compositions like

$$\texttt{let}\ \mathit{lengapp}\ l_1\ l_2 = \mathit{length}\ (\mathit{append}\ l_1\ l_2)$$

```
let length x = case x with
  cons head tail →
    1 + (length tail)
  nil → 0
```

```
let append l1 l2 = case l1 with
  cons head tail →
    cons head (append tail l2)
  nil → l2
```

Intuitively, these techniques process in three steps. First, they *expose* constructors to functions (unfolding).

$$\begin{array}{l}\texttt{let}\ \mathit{lengapp}\ l_1\ l_2 = \texttt{case}\ l_1\ \texttt{with}\\ \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow\\ \quad\quad \mathit{length}\ (\mathit{cons}\ \mathit{head}\ (\mathit{append}\ \mathit{tail}\ l_2))\\ \quad \mathit{nil} \rightarrow \mathit{length}\ l_2\end{array}$$

Next, they apply a kind of *partial evaluation* to these terms (application to constructors), that carries out the elimination of intermediate data structure.

$$\begin{array}{l}\texttt{let}\ \mathit{lengapp}\ l_1\ l_2 = \texttt{case}\ l_1\ \texttt{with}\\ \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow\\ \quad\quad 1 + (\mathit{length}\ (\mathit{append}\ \mathit{tail}\ l_2))\\ \quad \mathit{nil} \rightarrow \mathit{length}\ l_2\end{array}$$

Finally, recursive *function calls* could be reintroduced[1] (folding).

$$\begin{array}{l}\texttt{let}\ \mathit{lengapp}\ l_1\ l_2 = \texttt{case}\ l_1\ \texttt{with}\\ \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow\\ \quad\quad 1 + (\mathit{lengapp}\ \mathit{tail}\ l_2)\\ \quad \mathit{nil} \rightarrow \mathit{length}\ l_2\end{array}$$

In the resulting *lengapp* function definition, the *conses* of the intermediate list have been removed.

Even if each technique is particular in its algorithm implementation (rewriting rule system [25], `foldr/build` elimination rule [9], fold normalization [23], hylomorphisms fusion [10]), they are more or less based on these three steps [5].

1.2 Unresolved cases

In spite of several improvements and refinements of functional techniques, the deforestation of a class of program still remains impossible. This concerns the composition of programs that construct (part of) their result in an accumulative parameter.

As a striking example, let us consider the function *rev* which reverses a list. In the following definition, parameter y is initialized with the value *nil*:

$$\begin{array}{l}\texttt{let}\ \mathit{rev}\ x\ y = \texttt{case}\ x\ \texttt{with}\\ \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow\\ \quad\quad \mathit{rev}\ \mathit{tail}\ (\mathit{cons}\ \mathit{head}\ y)\\ \quad \mathit{nil} \rightarrow y\end{array}$$

The classical functional composition of this function with itself leads to the function definition $\texttt{let}\ \mathit{revrev}\ x\ y\ z = \mathit{rev}\ (\mathit{rev}\ x\ y)\ z$, where the list built by the inner *rev* is the

[1] Depending on the deforestation method, this step is implicit or not in the process.

intermediate data structure consumed by the outer *rev*. As far as we know, any general[2] existing deforestation method allows this composition to be transformed in a program that solely constructs the final list (x itself). Indeed, applying the previously presented three steps to this example leads to:

$$\begin{array}{l}\texttt{let } \mathit{revrev}\ x\ y\ z\ = \texttt{case } x \texttt{ with}\\ \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow\\ \quad\quad \mathit{revrev}\ \mathit{tail}\ (\mathit{cons}\ \mathit{head}\ y)\ z\\ \quad \mathit{nil} \rightarrow \mathit{rev}\ y\ z\end{array}$$

During the transformation process, the partial evaluation step has never been applied, so the intermediate list is still constructed in *revrev* function. The only difference with respect to the classical function composition is that the outer *rev* is now applied as soon as the inner inverted list is constructed (in y): instead of applying *rev* to the result of the first *rev* call, it is applied to the y accumulative parameter when it contains the whole inverted list.

Nevertheless, translating this example into attribute grammars, our deforestation method, namely the *symbolic composition*, produces a new attribute grammar that no more constructs the intermediate list. This attribute grammar could be translated into the following function definitions:

$$\textbf{(1)}\qquad \begin{array}{ll}\multicolumn{2}{c}{\texttt{let } \mathit{revrev}\ x\ y\ z = \mathit{f2}\ x\ (\mathit{rev}\ y\ (\mathit{f1}\ x\ z))}\\ \texttt{let } \mathit{f2}\ x\ t = \texttt{case } x \texttt{ with} & \texttt{let } \mathit{f1}\ x\ z\ = \texttt{case } x \texttt{ with}\\ \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow & \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow\\ \quad\quad \mathit{f2}\ \mathit{tail}\ t & \quad\quad \mathit{cons}\ \mathit{head}\ (\mathit{f1}\ \mathit{tail}\ z)\\ \quad \mathit{nil} \rightarrow t & \quad \mathit{nil} \rightarrow z\end{array}$$

The intermediate list has been completely discarded in these functions, even if a useless traversal (*f2*) of the tree remains. In fact, in the particular case of attribute grammars, a copy rule elimination could even discard this traversal [21].

1.3 Accumulative parameter deforestation

The generalization of this example allows us to outline the counterpart of symbolic composition in terms of classical functional methods. Our proposition is to add a kind of eureka step to classical functional methods.

In the previous example, this eureka step is just to set $\mathit{revrev}\ x\ y\ z = \mathit{f2}\ x\ (\mathit{rev}\ y\ (\mathit{f1}\ x\ z))$, where *f1* do a pattern-matching on the same x as *f2*. Then, the *revrev* definition obtained by classical methods could be rewrote into the following **(2)** definition and applying the

[2]This particular example could be deforested with a dedicated method [23] that cannot be applied, for instance, to *rev* (*flat* t l) h (cf. section 2).

partial evaluation step leads to **(3)**:

$$\textbf{(2)} \qquad \begin{array}{l} \texttt{let}\ revrev\ x\ y\ z\ \equiv\ f2\ x\ (rev\ y\ (f1\ x\ z))\ = \texttt{case}\ x\ \texttt{with} \\ \quad cons\ head\ tail \rightarrow f2\ tail\ (rev\ (cons\ head\ y)\ (f1\ tail\ z)) \\ \quad nil \rightarrow rev\ y\ z \end{array}$$

$$\textbf{(3)} \qquad \begin{array}{l} \texttt{let}\ revrev\ x\ y\ z\ \equiv\ f2\ x\ (rev\ y\ (f1\ x\ z))\ = \texttt{case}\ x\ \texttt{with} \\ \quad cons\ head\ tail \rightarrow f2\ tail\ (rev\ y\ (cons\ head\ (f1\ tail\ z))) \\ \quad nil \rightarrow rev\ y\ z \end{array}$$

Then, a generalized folding step on **(3)**, taking into account the fact that *f1* and *f2* pattern-match the same x, leads to previous deforested definitions **(1)**.

Now, let us try to explain how "the" eureka step could be deduced from the *revrev* function in classical deforestation method. The key point is that *rev* is applied to an accumulative parameter y in the *nil* case.

The first idea is to compute the reverse of the accumulative parameter y *while* it is constructed instead of *after* it is accumulated. To do so, a new accumulative parameter, always equal to $rev\ y\ z$, has to be added to the original function. We deduce a first approximation of the eureka step, which looks like $revrev\ x\ y\ z = f2\ x\ y\ z\ (rev\ y\ z)$. But this eureka step is wrong since, generally, the z parameter on the *nil* case is not equal to the one ahead of the list.

The second idea is then to notice that part of the *revrev* function computes the value of z, doing pattern-matching on the list x. So, this part could be extracted[3] into a separate function *f1* that does pattern-matching on the same list x. So, $revrev\ x\ y\ z = f2\ x\ y\ z\ (rev\ y\ (f1\ x\ y\ z))$ appears to be a correct eureka step and leads to the following function definitions:

$$\textbf{(4)} \qquad \begin{array}{ll} \multicolumn{2}{c}{\texttt{let}\ revrev\ x\ y\ z = f2\ x\ y\ z\ (rev\ y\ (f1\ x\ y\ z))} \\ \texttt{let}\ f2\ x\ y\ z\ t = \texttt{case}\ x\ \texttt{with} & \texttt{let}\ f1\ x\ y\ z\ = \texttt{case}\ x\ \texttt{with} \\ \quad cons\ head\ tail \rightarrow & \quad cons\ head\ tail \rightarrow \\ \qquad f2\ tail\ (\underline{cons\ head\ y})\ \underline{z}\ t & \qquad cons\ head\ (f1\ tail\ (\underline{cons\ head\ y})\ z) \\ \quad nil \rightarrow t & \quad nil \rightarrow z \end{array}$$

In *f1* and *f2*, a simple analysis shows that underlined computations are never used in the final results. The omission of the corresponding useless parameters leads to the deforested function definitions, *f1* and *f2*, as those in **(1)**. A same result could have been produced, directly applying such an analysis to *revrev* definition **(3)** in order to generate the eureka step deduced from symbolic composition **(1)**.

[3] By an inductive proof on the list data type.

2 Embedding the symbolic composition deforestation in a functional programming framework

In introduction, some ideas to achieve deforestation in accumulative parameters have been drafted for functional framework. To generalize these intuitions, a first formalization could be conjectured as follows.

Consider two functions[4], $\Psi\ p\ \overline{x}$ and $\Phi\ q\ \overline{y}$, pattern-matching their first parameter and their composition $\Phi\ (\Psi\ p\ \overline{x})\ \overline{y}$. If, after applying classical deforestation steps, some term $\Phi\ x_i\ \overline{(g\ \overline{z})}$ remains, then eureka step $\Phi\Psi\ p\ \overline{x}\ \overline{y} \equiv f2\ p\ \overline{x}\ \overline{y}\ (\Phi\ x_i\ \overline{(f1\ p\ \overline{x}\ \overline{y})})$ could be produced.

Another way to formalize these ideas consists in using their native formalism. Since they comes from attribute grammar techniques, especially from symbolic composition, we will define a translation FP-to-AG, from a functional program into its attribute grammar form, on which symbolic composition can be efficiently applied. Then, such deforested attribute grammars could be translated back into functional programs.

We have chosen this second presentation for the following reasons. Specificities of attribute grammars permit a uniform formalization, since they allow the same techniques to be applied for accumulative parameter as for simple deforestation. Indeed, previous work [5] has shown that attribute grammar deforestation and classical functional techniques are comparable in their effects for simplest functions (without accumulative parameters deforestation) and *attributes* attached to pattern variables allow results and parameters to be identically treated since all of them are defined by simple oriented equations, expressing both bottom-up (synthesized) and top-down (inherited) computations. Moreover, an attribute grammar specification can gather several computations (simple functions like *f1* and *f2*). Finally, this approach allows symbolic composition implementation to be reused.

All the process will be illustrated by the deforestation example of the composition[5] of *rev* with *flat*, where the function *flat* computes the list of the leaves of a given binary tree (cf. Fig. 1). The list constructed by *flat* before to be consumed by *rev* is then the intermediate data structure to be eliminated.

[4] $\overline{x}$ stands for a n-uple $x_1, \ldots, x_n$.

[5] We chose this example instead of *revrev* for explanatory purpose, because it involves two different functions and then avoids name confusions.

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

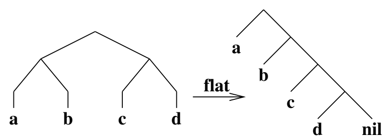

Figure 1: Function definition for *flat*

$$
\begin{array}{lcll}
\mathit{prog} & ::= & \overline{\mathit{def}} & \\
\mathit{def} & ::= & \texttt{let}\ f\ \overline{x} = \mathit{exp} & \\
 & | & \texttt{let}\ f\ \overline{x} = \texttt{case}\ x_k\ \texttt{with}\ \{\mathit{pat} \rightarrow \mathit{exp}\}^{+} & \\
\mathit{pat} & ::= & c\ \overline{x} & \\
\mathit{exp} & ::= & \mathit{Constant} & \\
 & | & x \in \mathit{Variables} & \\
 & | & g\ \overline{\mathit{exp}} \quad \mathit{Function\ call} &
\end{array}
$$

Figure 2: Functional language

$$
\begin{array}{lcl}
\mathit{block} & ::= & \texttt{aglet}\ f = \{f\ \overline{x} \rightarrow \overline{\mathit{semrule}}\}\{\mathit{pat} \rightarrow \overline{\mathit{semrule}}\}^{*} \\
\mathit{semrule} & ::= & \mathit{occ} = \mathit{exp} \\
\mathit{occ} & ::= & x.a \mid f.\mathit{result} \\
\mathit{exp} & ::= & \mathit{Constant} \\
 & | & y.b \in \mathit{Attribute\ occurrences} \\
 & | & x \in \mathit{Variables} \\
 & | & g\ \overline{\mathit{exp}} \quad \mathit{Attribute\ grammar\ call}
\end{array}
$$

Figure 3: Attribute grammar notation

2.1 Language syntaxes

To present the basic steps of our transformations in a simple and clear way, we deliberately restrict ourselves to a sub-class of first order functional programs with the syntax presented in Fig. 2. Notice that nested pattern-matching are not allowed, but it is easy to split them in several separated functions. Moreover, the statements *if-then-else* can be taken into account with *Dynamic Attribute Grammars* [20].

To bring our attribute grammar notation, presented in Fig. 3, closer to functional specifications, algebraic type definitions will be used instead of classical context free grammars [2, 7]. This notation is not the classical one, but is a minimal form for explanatory purpose. Thus, a grammar production is represented as a data-type constructor followed by its parameter variables, that is, a pattern (for example: *cons head tail*).

Moreover, a characteristic feature of attribute grammars is to distinguish two kinds of attributes: the *synthesized* ones are computed bottom-up over the structure and the *inherited* ones are computed top-down. Since our transformations will consider type-checked functional programs as input, this induces informations about the generated attribute grammars. Thus, the sort and the type of attribute are directly deduced from the type-checked input program and could be implicit.

```
aglet rev =
  rev x h →
    rev.result = x.rev
    x.h = h
  cons head tail →
    cons.rev = tail.rev
    tail.h = cons head cons.h
  nil →
    nil.rev = nil.h
```

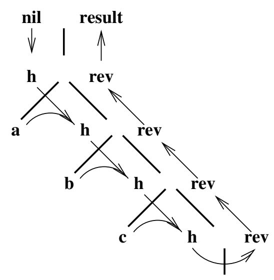

Figure 4: Attribute grammar *rev*

Furthermore, the notion of *attribute grammar profile* is introduced (in Fig. 3, $f\ \overline{x}$ is the profile of f). It represents how to call the attribute grammar and allows result and arguments to be specified.

The occurrence of an attribute a on a pattern variable x is noted $x.a$, even if this pattern variable is the constructor of the current pattern itself. For instance, according to the syntax in Fig. 3, the function *rev* could be specified by the attribute grammar in Fig. 4. This figure contains also an intuitive illustration for the application *rev* (*cons a* (*cons b* (*cons c nil*))) *nil*.

With this notation, the name *rev* stands all at the same time for the attribute grammar, for the profile constructor and for a synthesized attribute. Variable x, the list to be reversed, is the pattern-matched argument and h is the parameter. The attribute *result* is the only synthesized of the profile. Variable x, and all pattern-matched (sub-)list, have two attributes: *rev* synthesized and h inherited. Each oriented equation defines an occurrence attribute.

2.2 The Translation FP-to-AG

The intuitive idea for the translation FP-to-AG, from a functional program into its attribute grammar notation, 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. FP-to-AG is decomposed into a *preliminary transformation* and a *profile symbolic evaluation*.

Preliminary Transformation The aim of the preliminary transformation, presented in Fig. 6, 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*). Each

$\stackrel{def}{=}$	:	local definition in an algorithm
$x.a = exp$	:	semantic rule defining $x.a$
$[x := y]$	:	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.

Figure 5: Notation used in further definitions and transformations

$$\frac{\forall i \quad \stackrel{exp}{\vdash} a_i \Rightarrow b_i \quad ; \quad f \textit{ is a function name}}{\stackrel{exp}{\vdash} (f\ \overline{a}) \Rightarrow (f\ \overline{b}).result} \qquad (App)$$

$$\frac{\stackrel{exp}{\vdash} e \Rightarrow e'}{f, \{x_j\}_{j \neq k}, x_k \stackrel{pat}{\vdash} c\ \overline{y} \rightarrow e \Rightarrow c\ \overline{y} \rightarrow c.f = e'[x_k := c][x_j := c.x_j]_{\forall j \neq k}} \qquad (Pattern)$$

$$\frac{\begin{array}{c} \forall i \quad f, \{x_j\}_{j \neq k}, x_k \stackrel{pat}{\vdash} p_i \rightarrow e_i \Rightarrow \Pi_i \\ \overline{\Pi} \stackrel{def}{=} \left(\begin{array}{l} f\ \overline{x} \rightarrow \\ \quad f.result = x_k.f \\ \quad x_k.x_j = x_j \quad {\scriptstyle (\forall j \neq k)} \end{array} \right) \cup \Pi_i \end{array}}{\stackrel{let}{\vdash} \texttt{let } f\ \overline{x} = \texttt{case } x_k \texttt{ with } \overline{p \rightarrow e} \Rightarrow \texttt{aglet } f = \overline{\Pi}} \qquad (Let)$$

$$\frac{\stackrel{exp}{\vdash} e \Rightarrow e'}{\stackrel{let}{\vdash} \texttt{let } f\ \overline{x} = e \Rightarrow \texttt{aglet } f = f\ \overline{x} \rightarrow f.result = e'} \qquad (Let')$$

$\stackrel{exp}{\vdash} e \Rightarrow e'$	means that the equation e is translated into the equation e'.
$env \stackrel{pat}{\vdash} p \rightarrow e \Rightarrow p \rightarrow \mathcal{R}$	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*.
$\stackrel{let}{\vdash} \mathcal{D} \Rightarrow \mathcal{B}$	means that the function definition $\mathcal{D}$ is translated into the *block* $\mathcal{B}$.

Figure 6: Preliminary transformation

```
aglet flat =
  flat t l →
    flat.result = t.flat
    t.l = l
  node left right →
    node.flat = (flat left (flat right node.l).result).result
  leaf n →
    leaf.flat = cons n leaf.l
```

Figure 7: The function *flat* after the preliminary transformation

supplementary argument yields a semantic rule defining an inherited attribute attached to the pattern-matched variable (rule *Let*). Each function call $(f\ \overline{a})$ is translated into a dotted notation $(f\ \overline{b}).result$ (rule *App*). This rule distinguishes between function and type constructor calls[6]. Each expression appearing in a pattern is transformed into a single semantic rule which defines the synthesized attribute computing the result (rule *App*). This induces some renaming (rule *Pattern*).

The application of the preliminary transformation to the function *flat* (Fig. 1) leads to the result shown in Fig. 7.

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. Fig. 3) which contains the profile of the function and its related patterns. But explicit recursive calls have been translated into the form $(f\ \overline{a}).result$. Now, these expressions have to be transformed into a set of semantic rules, breaking explicit recursions by attribute naming and attachment to pattern variables. Then, these semantic rules will implicitly define the recursion *à la* attribute grammar. This transformation is achieved by the profile symbolic evaluation presented in Fig. 8.

Everywhere an expression $(f\ \overline{a}).result$ 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 predicate *Check* 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* forbids some non-linear terms such as $g\ (f\ x\ 1)\ (f\ x\ 2)$. Moreover, in a first approach, terms like $(x.a).b$ are not allowed but they will be treated in section 2.3. Wherever $Check(c, f, \Sigma)$ is not verified, the expression $(f\ \overline{a}).result$ is simply rewritten in the function call $(f\ a)$.

The application[7] of the profile symbolic evaluation on this semantic rule is presented in Fig. 9. Finally, complete application of the profile symbolic evaluation for the function

[6]This distinction is performed from type information of the input functional program.

[7]Underlined <u>*terms*</u> show where the rule is being applied.

$$
\frac{
\begin{pmatrix} f\ \overline{x} \rightarrow \\ \quad f.result = \varphi \\ \quad \Sigma_f \end{pmatrix} \in \mathcal{P}
\qquad
\begin{array}{l} \sigma \stackrel{def}{=} [x_i := a_i] \\ \Sigma \stackrel{def}{=} \begin{cases} u = \mathcal{E}[\sigma(\varphi)] \\ \sigma(\Sigma_f) \\ \Sigma_{aux} \end{cases} \end{array}
\qquad
Check(c, f, \Sigma)
}{
\mathcal{P} \vdash \begin{pmatrix} c\ \overline{y} \rightarrow \\ \quad u = \mathcal{E}[(f\ \overline{a}).result] \\ \quad \Sigma_{aux} \end{pmatrix} \Rightarrow \begin{pmatrix} c\ \overline{y} \rightarrow \\ \Sigma \end{pmatrix}
}
\quad (PSE)
$$

$\mathcal{P} \vdash p \rightarrow \Sigma_1 \ \Rightarrow\ p \rightarrow \Sigma_2$ means that in the program $\mathcal{P}$ the set of equations Σ_1 of a pattern p is transformed into Σ_2.

Figure 8: Profile symbolic evaluation

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

Figure 9: Example of *PSE* application for the pattern *node left right*

flat leads to the well-formed attribute grammar given in Fig. 10. This figure gives also an illustrative example of *flat* application on the tree *node* (*leaf a*) (*node* (*leaf b*) (*leaf c*)).

The same algorithm applied to the function *rev* (section 1.2) yields the attribute grammar in Fig. 4. Then, the successive application of preliminary transformation and profile symbolic evaluation to an input functional program leads to a real attribute grammar. This is the translation FP-to-AG.

2.3 Symbolic Composition

It is now possible to apply attribute grammars deforestation methods to functional programs translated by FP-to-AG. To be able to present our symbolic composition, we first present a natural extension of profile symbolic evaluation which 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

$$\begin{array}{l} \texttt{aglet}\ \mathit{flat} = \\ \quad \mathit{flat}\ t\ l \rightarrow \\ \quad\quad \mathit{flat.result} = t.\mathit{flat} \\ \quad\quad t.l = l \\ \quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\ \quad\quad \mathit{node.flat} = \mathit{left.flat} \\ \quad\quad \mathit{left}.l = \mathit{right.flat} \\ \quad\quad \mathit{right}.l = l \\ \quad \mathit{leaf}\ n \rightarrow \\ \quad\quad \mathit{leaf.flat} = \mathit{cons}\ n\ \mathit{leaf}.l \end{array}$$

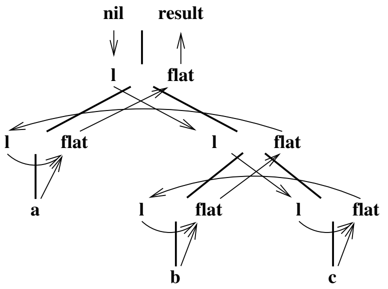

Figure 10: Attribute grammar produced by FP-to-AG from function *flat*

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

Figure 11: Symbolic Evaluation

more *blocks* of attribute grammars rather than complete attribute grammars. Furthermore, the expressions of the form $(x.a).b$ previously avoided (cf. predicate $Check$ in PSE), will be temporarily authorized for the symbolic composition process.

Symbolic Evaluation Profile symbolic evaluation can be generalized into a new symbolic evaluation, presented in Fig. 11, which performs both profile symbolic evaluation and partial evaluation on finite terms. The idea of this transformation is to recursively project semantic rules on finite terms and to eliminate intermediate attributes that are defined and used in the produced semantic rules.

To illustrate the use of symbolic evaluation as partial evaluation, consider the term $\texttt{let}\ g\ z = \mathit{rev}\ (\mathit{cons}\ a\ (\mathit{cons}\ b\ \mathit{nil}))\ z$. Applying FP-to-AG to this term yields the following attribute grammar:

$$\begin{array}{l} \texttt{aglet}\ g\ z \rightarrow \\ \quad g.\mathit{result} = (\mathit{cons}\ a\ (\mathit{cons}\ b\ \mathit{nil})).\mathit{rev} \\ \quad (\mathit{cons}\ a\ (\mathit{cons}\ b\ \mathit{nil})).h = z \end{array}$$

$$\frac{\begin{pmatrix} cons\ head\ tail \rightarrow \\ \quad cons.rev = tail.rev \\ \quad tail.h = cons\ head\ cons.h \end{pmatrix} \in \mathcal{P} \qquad \begin{array}{l} \sigma = [head := a][tail := cons\ b\ nil][cons.h = z] \\ \Sigma = \left\{ \begin{array}{l} g.result = (cons\ b\ nil).rev \\ (cons\ b\ nil).h = cons\ a\ z \end{array} \right. \\ Check(g, cons, \Sigma) \end{array}}{\mathcal{P} \vdash \quad g\ z \rightarrow \left\{ \begin{array}{l} g.result = (cons\ a\ (cons\ b\ nil)).rev \\ (cons\ a\ (cons\ b\ nil)).h = z \end{array} \right\} \quad \Rightarrow \quad g\ z \rightarrow \Sigma}$$

Figure 12: Example of *SE* application for *rev* (*cons a* (*cons b nil*)) *z*

Then, the symbolic evaluation (Fig. 11) could be applied on these terms. The first step of this application is presented in Fig. 12. Two other steps of this transformation lead to $g.result = (cons\ b\ (cons\ a\ z))$.

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

Composition Getting back to our example, let us consider the definition of the function *revflat* which flattens a tree and then reverses the obtained list.

$$\texttt{let}\ revflat\ t\ l\ h = rev\ (flat\ t\ l)\ h$$

Intuitively, in the context of attribute grammar notation, this composition involves the two sets of attributes $Att_{flat} = \{flat\ \ ,\ l\}$ and $Att_{rev} = \{rev\ \ ,\ h\}$.

More generally, consider an attribute grammar $\mathcal{F}$ (e.g., *flat*), producing an intermediate data structure to be consumed by another attribute grammar $\mathcal{G}$ (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 descriptional 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 (*Proj*) is presented in Fig. 13. Then, the application of the symbolic evaluation will eliminates the useless constructors.

However, a point remains undefined: how to find the application sites for the projection steps *Proj* ? As attended, the predicate *Check*, 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 predicate *Check* and from the definition of the function *revflat*, we obtained the blocks presented in Fig. 14 (first is for the *revflat* profile, and others correspond to the attribute grammars *flat* and *rev*). In the blocks building the intermediate data

$$\frac{a \in Att_{\mathcal{F}} \quad \overline{s} = Att\_S_{\mathcal{G}} \quad \overline{h} = Att\_H_{\mathcal{G}}}{Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash x.a = e \ \Rightarrow \begin{cases} (x.a).s = (e).s & \forall s \in \overline{s} \\ (e).h = (x.a).h & \forall h \in \overline{h} \end{cases}} \quad (Proj)$$

$Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash eq \Rightarrow \Sigma$	means that, while considering $\mathcal{G} \circ \mathcal{F}$, the equation eq is transformed into the set of equations Σ.
$Att\_S_{\mathcal{G}}$	is the set of synthesized attributes of $Att_{\mathcal{G}}$.
$Att\_H_{\mathcal{G}}$	is the set of inherited attributes of $Att_{\mathcal{G}}$.

Figure 13: Projection step

structure, the application sites for the projection step *Proj* are underlined, and a $*$ highlights the construction to be deforested.

Fig. 15 shows the projection step for the pattern *leaf n* and all applications of this steps yield the blocks in the left part of Fig. 16.

Now, symbolic evaluation could be applied on annotated site, performing the real deforestation. For the first annotated site, the predicate *Check* is not verified and then a function call is reintroduced ($rev\ l\ (t.l).h$). New attributes are created by renaming attributes $a.b$ into a_b (when $a \in Att_{\mathcal{F}}$ and $b \in Att_{\mathcal{G}}$). More precisely, $(x.a).b$ is transformed into $x.a\_b$.

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

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

Thus, for the function *revflat*, the symbolic composition leads to the deforested attribute grammar presented in the right part of Fig. 16, where four attributes have been generated. Producing a functional evaluator for this attribute grammar yields the functions[8] *revflat*, *f1* and *f2* presented in Fig. 17.

The function *f1*, corresponding to attributes l_h (its result) and $flat\_h$ (its argument), performs the construction of a list. The function *f2*, corresponding to attributes $flat\_rev$ (its result) and l_rev (its argument), only propagates its argument along the tree. Then, the second parameter in the call $f2\ t\ (rev\ l\ (f1\ t\ h))$ corresponds to the semantic rule $t.l\_rev = (rev\ l\ t.l\_h)$ in the profile of the attribute grammar. Indeed, since $t.l\_h$ stands for the call $(f1\ t\ h)$ and since $t.l\_rev$ corresponds to the second argument of *f2*, the later stands for $rev\ l\ (f1\ t\ h)$.

The intermediate list is *no more constructed* and *revflat* is deforested.

Current limitations 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

[8]Functions *f1* and *f1* respectively correspond to the traversal (*passes*) determined by the attribute grammar evaluator generator.

$$\begin{array}{l} \textit{revflat}\ t\ l\ h \rightarrow \\ \quad \textit{revflat.result} = (t.\textit{flat}).\textit{rev} \\ \quad (t.\textit{flat}).h = h \\ \quad \underline{t.l = l} \end{array}$$

$$\begin{array}{l} \textit{node left right} \rightarrow \\ \quad \underline{\textit{node.flat} = \textit{left.flat}} \\ \quad \underline{\textit{left.l} = \textit{right.flat}} \\ \quad \underline{\textit{right.l} = \textit{node.l}} \\ \textit{leaf}\ n \rightarrow \\ \quad \underline{\textit{leaf.flat} = \textit{cons}\ n\ \textit{leaf.l}} \quad * \end{array} \qquad \begin{array}{l} \textit{cons head tail} \rightarrow \\ \quad \textit{cons.rev} = \textit{tail.rev} \\ \quad \textit{tail.h} = \textit{cons head cons.h} \\ \textit{nil} \rightarrow \\ \quad \textit{nil.rev} = \textit{nil.h} \end{array}$$

Figure 14: Blocks for *revflat* before projection steps

$$\frac{\textit{flat} \in Att_{\textit{flat}} \qquad \begin{array}{l} \overline{s} = Att\_S_{rev} = \{rev\} \\ \overline{h} = Att\_H_{rev} = \{h\} \end{array}}{Att_{rev}, Att_{\textit{flat}} \vdash \textit{leaf.flat} = \textit{cons}\ n\ \textit{leaf.l} \Rightarrow \left\{ \begin{array}{l} (\textit{leaf.flat}).rev = (\textit{cons}\ n\ \textit{leaf.l}).rev \\ (\textit{cons}\ n\ \textit{leaf.l}).h = (\textit{leaf.flat}).h \end{array} \right.} \quad (Proj)$$

Figure 15: Example of *Proj* application for the pattern *leaf n*

$$\begin{array}{l} \textit{revflat}\ t\ l\ h \rightarrow \\ \quad \textit{revflat.result} = (t.\textit{flat}).\textit{rev} \\ \quad (t.\textit{flat}).h = h \\ \quad \left.\begin{array}{l} (t.l).rev = (l).rev \\ (l).h = (t.l).h \end{array}\right\} \text{SE site but } \neg Check \\ \textit{node left right} \rightarrow \\ \quad (\textit{node.flat}).rev = (\textit{left.flat}).rev \\ \quad (\textit{left.flat}).h = (\textit{node.flat}).h \\ \quad (\textit{left.l}).rev = (\textit{right.flat}).rev \\ \quad (\textit{right.flat}).h = (\textit{left.l}).h \\ \quad (\textit{right.l}).rev = (\textit{node.l}).rev \\ \quad (\textit{node.l}).h = (\textit{right.l}).h \\ \textit{leaf}\ n \rightarrow \\ \quad \left.\begin{array}{l} (\textit{leaf.flat}).rev = (\textit{cons}\ n\ \textit{leaf.l}).rev \\ (\textit{cons}\ n\ \textit{leaf.l}).h = (\textit{leaf.flat}).h \end{array}\right\} \text{SE site} \end{array} \qquad \begin{array}{l} \texttt{aglet}\ \textit{revflat} = \\ \quad \textit{revflat}\ t\ l\ h \rightarrow \\ \quad\quad \textit{revflat.result} = t.\textit{flat\_rev} \\ \quad\quad t.\textit{flat\_h} = h \\ \quad\quad t.l\_rev = (rev\ l\ t.l\_h) \\ \quad \textit{node left right} \rightarrow \\ \quad\quad \textit{node.flat\_rev} = \textit{left.flat\_rev} \\ \quad\quad \textit{left.flat\_h} = \textit{node.flat\_h} \\ \quad\quad \textit{left.l\_rev} = \textit{right.flat\_rev} \\ \quad\quad \textit{right.flat\_h} = \textit{left.l\_h} \\ \quad\quad \textit{right.l\_rev} = \textit{node.l\_rev} \\ \quad\quad \textit{node.l\_h} = \textit{right.l\_h} \\ \quad \textit{leaf}\ n \rightarrow \\ \quad\quad \textit{leaf.flat\_rev} = \textit{leaf.l\_rev} \\ \quad\quad \textit{leaf.l\_h} = \textit{cons}\ n\ \textit{leaf.flat\_h} \end{array}$$

Figure 16: Attribute grammar *revflat* before and after symbolic evaluation and renaming

```
let revflat t l h = f2 t (rev l (f1 t h))
let f1 t h = case t with              let f2 t l = case t with
  node left right →                     node left right →
    f1 right (f1 left h)                  f2 left (f2 right l)
  leaf n → cons n h                     leaf n → l
```

Figure 17: Functions corresponding to the deforested attribute grammar *revflat*

introduced by Ganzinger and Giegerich about descriptional composition of attribute grammars [8].

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. This problem has to be related to the analysis, evoked in introduction **(4)**, allowing useless parameters to be omitted.

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 Conclusion

The goal of this paper is to show that it is possible to translate (interpret) the symbolic composition of attribute grammars in a functional framework.

More precisely, we show that symbolic composition, not only achieves similar effects as functional methods [5], but also successfully deforests in accumulative parameters. This result reinforces our conviction that the formalism of attribute grammars is simple and appropriate for this kind of transformations. The translation FP-to-AG (presented here in its simpler form) together with a classical reciprocal transformation should be viewed as formalization 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 symbolic composition. Furthermore, these limitations are not specific to our attribute grammar-based deforestation since they appear also in calculational systems, such as HYLO (see for example [17]).

Moreover, we extend the basic descriptional composition into a more powerful one: the symbolic composition. This extension allows it to be used as a partial evaluation. Furthermore, it could be applied to terms with function compositions, and not only to one composition of two distinct attribute grammars (attribute coupled grammars [8]) 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 paper.

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 in order to be able to specialize it in several contexts. Similar approaches are being studying in different programming paradigms (*polytypic* programming [11], *adaptive* programming [18]). We have also compared [4] these approaches with our genericity tools [15, 22, 21, 3], that have been implemented in our FNC-2 system [13]. 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.

References

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

[2] 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).

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

[4] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic programming by program composition (position paper). In *Workshop on Generic Programming*, Marstrand, Sweden, June 1998. conjunction with MPC'98.

[5] Etienne Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle*. PhD thesis, Université d'Orléans, 1998.

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

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

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

[9] 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.

[10] 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.

[11] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *24th ACM Symp. on Principles of Programming Languages*, 1997.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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, Linkoping, Sweden, 1996. Springer Verlag.

[19] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. Rapport de Recherche 2662, INRIA, October 1995.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399