Declarative Program Transformation: a Deforestation case-study

Loïc Correnson[1], Etienne Duris[2], Didier Parigot[1], and Gilles Roussel[3]

[1] INRIA-Rocquencourt - Domaine de Voluceau, BP 105 F-78153 Le Chesnay Cedex
{Loic.Correnson, Didier.Parigot}@inria.fr
[2] Cedric CNAM - 18, allée Jean Rostand F-91025 Evry Cedex
duris@iie.cnam.fr
[3] Institut Gaspard Monge - 5, bd Descartes F-77454 Marne-la-Vallée Cedex 2
roussel@univ-mlv.fr

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 structure constructions that occur when two functions are composed. Beyond classical compile time optimization, these transformations are undeniable tools for generic programming and software component specialization.
Despite various and numerous research works in this area, general transformation methods cannot deforest some non-trivial intermediate constructions. Actually, these recalcitrant structures are built inside *accumulating parameters* and then, they follow a construction scheme which is independent from the function scheme itself. Known deforestation methods are too much tied to fixed recursion schemes to be able to deforest these structures.
In this article, we show that a fully declarative approach of program transformation allows new deforestation sites to be detected and treated. We present the principle of the *symbolic composition*, based on the attribute grammar formalism, with an illustrative running example stemming from a typical problem of standard functional deforestations.
Key words. Program transformation, deforestation, attribute grammars, functional programming, partial evaluation.

1 Introduction

More than a decade ago, P. Wadler said *"Intermediate data-structures are both the basis and the bane of modular programming."* [29]. Indeed, if they allow functions to be composed, these data-structures 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.

The main motivation for deforestation transformations was, for a long time, compiler optimization. More recently, with the emergence of component-based software development, that requires both automatic software generation and component specialization, deforestation transformations find new interest again, just as partial evaluation or more generally high level source-to-source program transformations [6, 4].

Since 1990, different approaches have been developed in order to improve the efficiency of deforestation transformations. Wadler's algorithm [29], based on Burstall and Darlington *unfold/fold* strategy [1], has been improved and extended by several works [2, 12, 25, 27]. Another approach, the *deforestation in calculational form* [11, 26, 16, 28, 13], was based on algebraic notions. This latter aims at using categorial *functors* to capture both function and data-type patterns of recursion [18] to guide the deforestation process.

With a large degree of formalisms or notations, all these methods are able to deforest function compositions like the following:

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

$$\begin{array}{ll}
\texttt{let}\ \mathit{length}\ x = \texttt{case}\ x\ \texttt{with} & \texttt{let}\ \mathit{append}\ l_1\ l_2 = \texttt{case}\ l_1\ \texttt{with} \\
\quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow & \quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \\
\quad\quad 1 + (\mathit{length}\ \mathit{tail}) & \quad\quad \mathit{cons}\ \mathit{head}\ (\mathit{append}\ \mathit{tail}\ l_2) \\
\quad \mathit{nil} \rightarrow 0 & \quad \mathit{nil} \rightarrow l_2
\end{array}$$

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 or recognized[1] (folding).

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

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

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 or in its theoretical underlying formalism (rewriting rule system [29], `foldr/build` elimination rule [11], fold normalization [26], hylomorphisms fusion [13]), they are more or less based on these three steps [8].

Major characteristics of these methods are, on the one hand, to expose data-structure producers to data-structure consumers in order to find partial evaluation application sites and, on the other hand, to detect and drive this deforestation process by following a general recursion scheme[2], that comes from the function or the data structure recursive definitions.

Unfortunately, all these methods fail in the deforestation of a class of intermediate data structures. This concerns functions that build — part of — their result inside an accumulating parameter, that is, a data which is neither directly the result nor the pattern matched syntactic argument of the function, but an auxiliary argument. Given a pair of functions to be fused, when the producer function collects its result in an accumulating parameter, the constructors in that parameter are *protected* from the consumer. In this case, no deforestation normally occurs.

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

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

The classical functional composition of this function with itself leads to the function definition $\texttt{let}\ revrev\ x\ y\ z = rev\ (rev\ x\ y)\ z$, where the list built by the inner *rev* is the intermediate data structure consumed by the outer *rev*. As far as we know, no general[3] 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:

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

During the transformation process, the partial evaluation step has never been applied, so the intermediate list is still constructed in *revrev* function. The only

[2] This recursion scheme can be exploited very simply (syntactically) or more sophistically (using abstract categorial representations such as functors).

[3] This particular example could be deforested with a dedicated method [26] that cannot be applied, for instance, to $rev\ (\mathit{flat}\ t\ l)\ h$ (cf. section 2).

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 accumulating parameter when it contains the whole inverted list.

The reason of this problem is that, since the deforestation methods are based on function and data-structure recursion schemes, they are only able to guide a deforestation process which is strongly tied to these fixed recursion schemes. Thus, they cannot detect nor treat *unfaithful* constructions in accumulating parameters. This approach could be viewed as not declarative enough.

On the opposite, a distinctive feature of attribute grammars is that they are fully declarative specifications [19]. They allow a uniform representation of all computations (as well results as parameters) by simple oriented equations. More precisely, they distinguish synthesized (bottom-up computed) and inherited (top-down computed) attributes. By this way, constructions computed in inherited attributes (those of accumulating parameters) become accessible by a deforestation process. Actually, since the operational semantics of an attribute grammar rests on the resolution of an oriented equation system,[4] the recursion scheme of the represented function is no more explicitly required.

Translating our 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 then be translated back, by well-known techniques (by the way of functional evaluators [21, 15]) into the following function definitions:

$$\texttt{let}\ \mathit{revrev}\ x\ y\ z = \mathit{f2}\ x\ (\mathit{rev}\ y\ (\mathit{f1}\ x\ z))$$

(1)

```
let f2 x t = case x with          let f1 x z = case x with
  cons head tail →                   cons head tail →
    f2 tail t                          cons head (f1 tail z)
  nil → t                            nil → z
```

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 [23].

The remainder of this article is structured as follows. First, section 2 presents syntactic notations, both for functional and attribute grammar languages. Next, section 3 describes a translation from functional programs into equivalent attribute grammars. Essentially, it transforms accumulating parameters into inherited attributes and breaks explicit recursions into oriented equation systems. Then, section 4 shows the basic principles of the symbolic composition, detailed on an illustrative running example. In conclusion, we discuss related works and we sketch future — and current — works related to a generalized formalization of this technique and its implementation.

---

[4] The equation system constituted by all attribute occurrence definitions.

2 Language Syntaxes and Notations

Rather than the confusing example *revrev* of introduction[5], we will illustrate our transformations by the deforestation example of the composition 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. This example constitutes a typical problem since, as far as we know, no known deforestation method is able to deal with. Nevertheless, this example represents the class of functions where the data-structure producer builds its result with an accumulating variable.

$$
\begin{array}{l}
\texttt{let}\ \mathit{flat}\ t\ l = \texttt{case}\ t\ \texttt{with}\\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow\\
\qquad \mathit{flat}\ \mathit{left}\ (\mathit{flat}\ \mathit{right}\ l)\\
\quad \mathit{leaf}\ n \rightarrow \mathit{cons}\ n\ l
\end{array}
$$

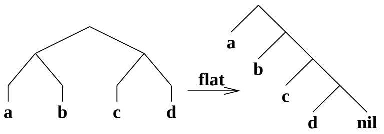

Fig. 1. Function definition for *flat*

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[6] presented in Fig. 2. Nested pattern-matching are not allowed, but are easy to split in several separated functions. Moreover, the statements *if-then-else* can be taken into account with *Dynamic Attribute Grammars* [22].

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 [3, 9, 8]. 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*).

[5] We prefer the *revflat* rather than the *revrev* example for explanatory purpose, because it involves two different functions and then avoids name confusions.

[6] Notation $\overline{x}$ stands for $x_1 \ldots x_n$.

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

Fig. 2. Functional language

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

Fig. 3. Attribute grammar notation

As previously said, a characteristic feature of attribute grammars is to distinguish two sorts 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 information about the generated attribute grammars. Thus, the sort and the type of attributes are directly deduced from the type-checked input program and could be implicit.

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[7]. 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

[7] In CFG terms, it plays the role of the left hand side (parent) of the production.

$$
\begin{array}{l}
\texttt{aglet}\ \mathit{rev} = \\
\quad \mathit{rev}\ x\ h \rightarrow \\
\qquad \mathit{rev.result} = x.\mathit{rev} \\
\qquad x.h = h \\
\quad \mathit{cons}\ \mathit{head}\ \mathit{tail} \rightarrow \\
\qquad \mathit{cons.rev} = \mathit{tail.rev} \\
\qquad \mathit{tail}.h = \mathit{cons}\ \mathit{head}\ \mathit{cons}.h \\
\quad \mathit{nil} \rightarrow \\
\qquad \mathit{nil.rev} = \mathit{nil}.h
\end{array}
$$

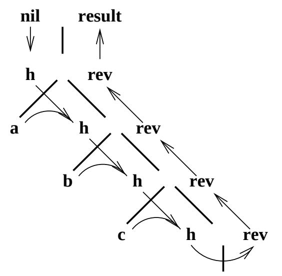

Fig. 4. Attribute grammar *rev*

pattern-matched (sub-)list, have two attributes: *rev* synthesized and h inherited. Each oriented equation defines an attribute for a given pattern variable.

3 Translation FP-to-AG

The intuitive idea of 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*.

These are notations used in further definitions and transformations.

$\stackrel{def}{=}$	: 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.

Preliminary Transformation The aim of the preliminary transformation, presented in Fig. 5, 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 profile and it stands for the expected result of the function (rule *Let'*). For function with `case`-statement the result is computed through attributes on the pattern-matched variable (rule *Let*): one is synthesized, named by the function name itself, and each supplementary argument of the function profile yields a semantic rule defining an inherited attribute attached to the pattern-matched variable.

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[8]. Thus, 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

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

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

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

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

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

Constants and variables are left unchanged by the transformation.

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

Fig. 5. Preliminary transformation

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

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

Fig. 6. Profile symbolic evaluation (*PSE*)

```
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
```

Fig. 7. The function *flat* after the preliminary transformation

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 (PSE) presented in Fig. 6.

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. Nevertheless, the predicate $Check_{PSE}$ ensures that the resulting attribute grammar is well formed. Essentially, it verifies that each attribute is defined once and only once. Then, in the context of well-defined input functional programs, $Check_{PSE}$ forbids non-linear terms such as $g\ (f\ y\ 1)\ (f\ y\ 2)$. Moreover, in a first approach, terms like $(x.a).b$ are not allowed but they will be treated in section 4, to detect composition sites. Finally, $Check_{PSE}$ prevents cyclic treatments with the condition $c \neq f$ and all these conditions allow FP-to-AG to terminate.

$$\frac{\begin{array}{c}\textit{In the pattern } c\ \overline{y} \to e\\ e \textit{ is linear for each } y_i\end{array} \qquad \begin{array}{c}\textit{No terms } (x.a).b\\ \textit{occurs in } \Sigma\end{array} \qquad c \neq f}{Check_{PSE}(c, f, \Sigma)}$$

Wherever $Check_{PSE}(c, f, \Sigma)$ is not verified, the expression $(f\ \overline{a}).result$ is simply rewritten in the function call $(f\ a)$.

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

Fig. 8. Example of PSE application for the pattern *node left right*

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

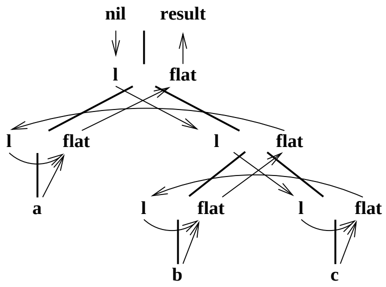

Fig. 9. Attribute grammar produced by FP-to-AG from function *flat*

The application[9] of the profile symbolic evaluation on the semantic rule for the *leaf* pattern is presented in Fig. 8. Finally, complete applications of the profile symbolic evaluation for the function *flat* leads to the well-formed attribute grammar given in Fig. 9. 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* (given in introduction) 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.

The cost of the preliminary transformation is linear with respect to the depth of input functional terms. Each function definition yields a profile block, with one semantic rule per each argument of the function. Furthermore, for each initial pattern case, a semantic rule defines an attribute occurrence that represents the value of the function in this case. These equations contain function calls that will be dismantled by the profile symbolic evaluation. The required number of application of this step is proportional to the number of recursive calls it contains. In this sense, the cost of FP-to-AG linearly depends on the size and the depth of the input functional program terms.

4 Symbolic Composition

It is now possible to apply attribute grammar deforestation methods to functional programs translated by FP-to-AG. Our technique, the symbolic composition, is based on the classical descriptional composition of two attribute grammars due to Ganzinger and Giegerich [10], but extends its application conditions and exploits the particular context stemming from translated functional programs. In order to describe 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 intermedi-

9 Underlined <u>*terms*</u> show where the rule is being applied.

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

Fig. 10. Symbolic Evaluation

ate transformations are more *blocks* of attribute grammars rather than complete attribute grammars. Furthermore, the expressions of the form $(x.a).b$, previously avoided (cf. predicate $Check_{PSE}$ in PSE), will be temporarily authorized by a similar *Check* predicate in the symbolic composition process.

Symbolic Evaluation Profile symbolic evaluation (PSE) can be generalized into a new symbolic evaluation (SE), presented in Fig. 10. This later 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 attribute occurrences that are defined and used in the produced semantic rules.

Indeed, rather than only project terms of the profile (function name) as in PSE, that is, on expressions $(f\ \overline{a}).result$, the symbolic evaluation SE will project terms related to each expression $(f\ \overline{a}).w$, were f stands as well for a type constructor as for an attribute grammar profile. Since these expressions could be coupled with inherited attribute occurrence definitions like $(f\ \overline{a}).h = \varphi_h$, corresponding to parameters of the function represented by w, these definitions must also be taken into account by the transformation.

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

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

Then, the symbolic evaluation (Fig. 10) could be applied on these terms. The first step of this application is presented in Fig. 11. Two other steps of this transformation lead to $g.result = (cons\ b\ (cons\ a\ z))$.

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

This generalization of the profile symbolic evaluation, into the symbolic evaluation used as a partial evaluation mechanism, implies that the complexity of

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

Fig. 11. Example of *SE* application for $rev\ (cons\ a\ (cons\ b\ nil))\ z$

this transformation directly relies on those of the treated terms. Practically, the number of symbolic evaluation applications must be arbitrary limited in order to prevent infinite loop, for instance in partial evaluation of an infinite list reversal. Nevertheless, at any stage of this process, a part of the computation has been symbolically performed.

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

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

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 of classical attribute grammars [10], 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. 12. Then, the application of the symbolic evaluation will eliminate the useless constructors.

From the complexity point of view, the projection step is essentially similar to the classical descriptional composition [10], that is, quadratic: the composition of two attribute grammars, respectively using n and m attributes, leads to $m * n$ attributes in the resulting attribute grammar, with as much semantic rules.

However, a point remains undefined: how to find the application sites for the projection steps *Proj* ? As attended, the predicate $Check_{PSE}$ is temporarily relaxed in *Check*, authorizing expressions like $(x.a).b$. 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. 13 (first is for the *revflat* profile, and others correspond to attribute grammars *flat* and *rev*). In the blocks building the intermediate data structure, potential application sites for the projection step *Proj* are underlined, and a $*$ highlights the construction to be deforested.

Fig. 14 shows the projection step for the pattern *leaf* and all applications of this steps yield the blocks in the left part of Fig. 15.

Now, symbolic evaluation could be tried on annotated sites, performing the real deforestation. The first annotated site is not a potential site for the symbolic evaluation application, since l is neither an attribute grammar (profile) call nor a type pattern constructor. In this case, as wherever *Check* is not verified, the computational context is reintroduced in the form of an attribute grammar (function) call ($rev\ l\ (t.l).h$). This functional call retrieval, together with linearity and distinct pattern ($c \neq f$) conditions of the *Check* predicate avoid infinite unfolding and ensure termination of the process (with the arbitrary limit for symbolic evaluation application mentioned in the partial evaluation discussion).

On the other hand, a symbolic evaluation step is successfully applied on the second annotated site, actually eliminating a *cons* construction. Finally, 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:

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

Thus, for the function *revflat*, the symbolic composition leads to the deforested attribute grammar presented in the right part of Fig. 15, where four attributes have been generated. Producing a functional evaluator for this attribute grammar yields the functions[10] *revflat*, *f1* and *f2* presented in Fig. 16.

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 $f2\ \ t\ (rev\ \ l\ (f1\ \ t\ h))$

[10] Functions *f1* and *f2* respectively correspond to the traversal (*passes*) determined by the attribute grammar evaluator generator.

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

$Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash eq \Rightarrow \Sigma$	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 $Att_{\mathcal{G}}$.

Fig. 12. Projection step

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

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

Fig. 13. Blocks for *revflat* before projection steps

$$
\frac{\mathit{flat} \in \mathit{Att}_{\mathit{flat}} \qquad \begin{array}{l} \overline{s} = \mathit{Att\_S}_{\mathit{rev}} = \{\mathit{rev}\} \\ \overline{h} = \mathit{Att\_H}_{\mathit{rev}} = \{h\} \end{array}}{\mathit{Att}_{\mathit{rev}}, \mathit{Att}_{\mathit{flat}} \vdash \mathit{leaf}.\mathit{flat} = \mathit{cons}\ n\ \mathit{leaf}.l \Rightarrow \left\{ \begin{array}{l} (\mathit{leaf}.\mathit{flat}).\mathit{rev} = (\mathit{cons}\ n\ \mathit{leaf}.l).\mathit{rev} \\ (\mathit{cons}\ n\ \mathit{leaf}.l).h = (\mathit{leaf}.\mathit{flat}).h \end{array} \right.}
$$

Fig. 14. Example of *Proj* application for the pattern *leaf n*

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. This achieves the presentation of our declarative deforestation methods on this typical example. Of course, this technique works equally well for simpler functions, without intermediate construction in accumulating parameters.

5 Conclusion

This paper shows that a fully declarative approach of program transformation could resolve a tenacious problem of deforestation: to deforest in accumulating parameters. The symbolic composition presented in this paper comes from a large comparison of deforestation techniques [8] and from the establishment that fixed recursion schemes, provided by data type specifications, are not flexible enough to catch all intermediate data structure constructions in function compositions. Several approaches attempted to abstract these recursion schemes in order to refine their manipulation, for instance by categorial representation [28, 13]. We were first surprised that these elaborate methods do not succeed in deforestations performed in the context of attribute grammar transformations. But two points differentiate them. First, attribute grammars are using fully declarative specifications, independently of any evaluation method, thanks to an operational semantics based on equation systems and dependencies res-

$revflat\ t\ l\ h \rightarrow$
$\quad revflat.result = (t.flat).rev$
$\quad (t.flat).h = h$
$\quad \left.\begin{array}{l}(t.l).rev = (l).rev \\ (l).h = (t.l).h\end{array}\right\}$ l is neither a function nor a constructor call
$node\ left\ right \rightarrow$
$\quad (node.flat).rev = (left.flat).rev$
$\quad (left.flat).h = (node.flat).h$
$\quad (left.l).rev = (right.flat).rev$
$\quad (right.flat).h = (left.l).h$
$\quad (right.l).rev = (node.l).rev$
$\quad (node.l).h = (right.l).h$
$leaf\ n \rightarrow$
$\quad \left.\begin{array}{l}(leaf.flat).rev = (cons\ n\ leaf.l).rev \\ (cons\ n\ leaf.l).h = (leaf.flat).h\end{array}\right\}$ SE site

`aglet` $revflat =$
$\quad revflat\ t\ l\ h \rightarrow$
$\quad\quad revflat.result = t.flat\_rev$
$\quad\quad t.flat\_h = h$
$\quad\quad t.l\_rev = (rev\ l\ t.l\_h)$
$\quad node\ left\ right \rightarrow$
$\quad\quad node.flat\_rev = left.flat\_rev$
$\quad\quad left.flat\_h = node.flat\_h$
$\quad\quad left.l\_rev = right.flat\_rev$
$\quad\quad right.flat\_h = left.l\_h$
$\quad\quad right.l\_rev = node.l\_rev$
$\quad\quad node.l\_h = right.l\_h$
$\quad leaf\ n \rightarrow$
$\quad\quad leaf.flat\_rev = leaf.l\_rev$
$\quad\quad leaf.l\_h = cons\ n\ leaf.flat\_h$

Fig. 15. 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`
$\quad node\ left\ right \rightarrow$
$\quad\quad f1\ right\ (f1\ left\ h)$
$\quad leaf\ n \rightarrow cons\ n\ h$

`let` $f2\ t\ l =$ `case` t `with`
$\quad node\ left\ right \rightarrow$
$\quad\quad f2\ left\ (f2\ right\ l)$
$\quad leaf\ n \rightarrow l$

Fig. 16. Functions corresponding to the deforested attribute grammar *revflat*

olution. Next, this declarative approach led them to use inherited attributes instead of supplementary arguments in order to specify top-down propagations or computations; this allows all computations — particularly intermediate data structure constructions — to be uniformly specified, and then, uniformly treated by transformations. This reinforces our conviction that the declarative formalism of attribute grammars is simple and appropriate for this kind of transformations.

Moreover, symbolic composition extends the descriptional composition: first, it could now be used as a partial evaluation mechanism and next, it could be applied to terms with function compositions, and not only to a sole composition of two distinct attribute grammars (attribute coupled grammars [10]) that are isolated of all context. For the attribute grammars community, this stands as the main contribution of this paper.

Nevertheless, as we wanted the presentation in this paper to be intuitive and convincing, accepted programs were limited by attribute grammar restrictions. For instance, non-linear terms, forbidden by the *Check* predicates, or higher order specifications are not addressed in this presentation for technical reasons due to the attribute grammar formalism. We now have formalized a complete system that includes and extends both symbolic composition and the declarative essence of attribute grammar formalism. *Equational se-*

mantics, fully detailed in [7], is able to encode an abstract representation of the operational semantics of a program. It supports simple transformations that could be combined into more complex ones. Its prototype implementation, `EQS`, is available and performs deforestation and partial evaluation (at `http://www-rocq.inria.fr/~correnso/agdoc/index.html`). Coupled with a FP-to-EQS translation, similar to FP-to-AG, `EQS` is able to deforest higher order functional programs, even authorizing some non-linear terms. Since `EQS` formalization is highly theoretical and language independent, the method and the transformation pipeline we have presented in this paper could be viewed as an intuitive presentation of these current — and future — works.

Finally, these works are involved in a more general study addressing genericity and reusability problems. The goal is to provide a set of high level transformational tools, able to abstract a given program and then to specialize it for several distinct contexts. We have compared [6] some attribute grammars tools [17, 24, 23, 5] with similar approaches in different programming paradigms (*polytypic* programming [14], *adaptive* programming [20]). Again, it appears in this context that declarative aspects of attribute grammars bring them particularly suitable for program transformations and that they should be viewed more as an abstract representation of a specification than as a programming language.

References

1. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
2. W. N. Chin and S. C. Khoo. Better consumers for deforestation. In *Prog. Lang.: Impl., Logic and Programs (PLILP'95)*, LNCS # 982, Springer-Verlag, 1995.
3. L. M. Chirica and D. F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979.
4. C. Consel. Program adaptation based on program specialization. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation. PEPM'99*, San Antonio, Texas, January 1999. ACM press.
5. L. Correnson. Généricité dans les grammaires attribuées. Rapport de stage d'option, École Polytechnique, 1996.
6. L. Correnson, E. Duris, D. Parigot, and G. Roussel. Generic programming by program composition (position paper). In *Workshop on Generic Programming (WGP'98)*, Marstrand, Sweden, June 1998.
7. L. Correnson, E. Duris, D. Parigot and G. Roussel. Equational Semantics. In *International Static Anaysis Symposium*, Veneszia, Italy, September 1999.
8. E. Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle*. PhD thesis, Université d'Orléans, October 1998.
9. H. Ganzinger, R. Giegerich, and M. Vach. MARVIN: a tool for applicative and modular compiler specifications. Forschungsbericht 220, Fachbereich Informatik, University Dortmund, July 1986.
10. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.
11. A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Functional Programming and Computer Architecture (FPCA'93)*, Copenhagen, Denmark, June 1993. ACM Press.

12. G. W. Hamilton. Higher order deforestation. In *Prog. Lang.: Impl., Logics and Programs (PLILP'96)*, LNCS # 1140, Aachen, September 1996. Springer-Verlag.
13. Z. Hu, H. Iwasaki, and M. Takeishi. Deriving structural hylomorphisms from recursive definitions. In *International Conference on Functional Programming (ICFP'96)*, Philadelphia, May 1996. ACM Press.
14. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Principles of Programming Languages (POPL'97)*, January 1997. ACM Press.
15. T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming and Computer Architecture (FPCA'87)*, LNCS # 274, Portland, September 1987. Springer-Verlag.
16. J. Launchbury and T. Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Functional Programming Languages and Computer Architecture (FPCA'95)*, La Jolla, CA, 1995. ACM Press.
17. C. Le Bellec, M. Jourdan, D. Parigot, and G. Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In *Prog. Lang.: Impl., Logic and Programs (PLILP '93)*, LNCS # 714, Tallinn, August 1993. Springer-Verlag.
18. E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming and Computer Architecture (FPCA'91)*, LNCS # 523, Cambridge, September 1991. Springer-Verlag.
19. J. Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
20. J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A new approach to compiling adaptive programs. In *European Symposium on Programming (ESOP'96)*, Linkoping, Sweden, 1996. Springer Verlag.
21. D. Parigot, E. Duris, G. Roussel, and M. Jourdan. Attribute grammars: a declarative functional language. Research Report 2662, INRIA, October 1995.
22. D. Parigot, G. Roussel, M. Jourdan, and E. Duris. Dynamic Attribute Grammars. In *Prog. Lang.: Impl., Logics and Programs (PLILP'96)*, LNCS # 1140, Aachen, September 1996. Springer-Verlag.
23. G. Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Université de Paris 6, March 1994.
24. G. Roussel, D. Parigot, and M. Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In *Compiler Construction (CC' 94)*, LNCS # 786, Edinburgh, April 1994. Springer-Verlag.
25. H. Seidl and M. H. Sørensen. Constraints to stop deforestation. *Science of Computer Programming*, 32(1-3):73-107, September 1998.
26. T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming and Computer Architecture (FPCA'93)*, Copenhagen, Denmark, June 1993. ACM Press.
27. M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *European Symposium on Programming (ESOP'94)*, LNCS # 788, Springer-Verlag, 1994.
28. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Functional Programming Languages and Computer Architecture (FPCA'95)*, La Jolla, CA, 1995. ACM Press.
29. P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming (ESOP '88)*, LNCS # 300, Nancy, March 1988. Springer-Verlag.