Composition Symbolique

Loïc Correnson & Etienne Duris
& Didier Parigot & Gilles Roussel[1]

INRIA, Domaine de Voluceau, Rocquencourt,
BP 105, 78153 Le Chesnay Cedex, France
{Loic.Correnson,Etienne.Duris,Didier.Parigot}@inria.fr
1: Université de Marne-la-Vallée,
2, allée du Promontoire, 93166 Noisy-le-Grand, France,
roussel@univ-mlv.fr

Résumé

La *déforestation* d'un programme fonctionnel est une transformation qui consiste à éliminer la construction des structures intermédiaires qui sont introduites par les compositions de fonctions. La *composition descriptionnelle*, initialement introduite par Ganzinger et Giegerich, est une méthode de déforestation spécifique, qui s'applique à la composition de deux grammaires attribuées.

Cet article propose une nouvelle technique de déforestation, appelée *composition symbolique*, qui est une extension et une amélioration de la composition descriptionnelle. En traduisant automatiquement un programme fonctionnel en une grammaire attribuée équivalente, il est possible de lui appliquer la composition symbolique, et de traduire le résultat en un programme fonctionnel (par exemple, en utilisant la transformation de Johnsson). On obtient alors une transformation source à source de programmes fonctionnels.

La méthode de déforestation ainsi obtenue donne de meilleurs résultats que les méthodes fonctionnelles existantes. La composition symbolique, complètement dédiée au caractère déclaratif et descriptionnel des grammaires attribuées est intrinsèquement plus puissante que les transformations basées sur les notions catégorielles, dont les schémas de récursions sont figés par des foncteurs.

Ces résultats confirment que la notation des grammaires attribuées est une représentation intermédiaire simple et particulièrement adaptée aux transformations de programmes.

1. Introduction

Les structures de données intermédiaires sont la base de la programmation modulaire. Elles en sont également le principal inconvénient, puisqu'elles coûtent cher en allocation et récupération de place mémoire. Les transformations de *déforestation* ont été introduites pour remédier à ce problème. Elles consistent en effet à fusionner deux parties modulaires et composables d'un programme en une seule, dans laquelle les constructions des structures intermédiaires ont été éliminées. Une des premières approches de la déforestation fut celle de Wadler [27], s'appuyant sur les transformations "*fold and unfold*".

Une autre approche, basée sur des notions algébriques, est connue sous le nom de déforestation *in calculational form* [9, 25, 15, 26, 10]. L'idée de cette approche est de capturer en même temps le schéma de récursion des fonctions et celui des types de données [18], afin de diriger les transformations par ces schémas de récursion.

Les grammaires attribuées [14, 20] sont des spécifications déclaratives et dirigées par la structure. Pour chaque schéma de récursion d'un type de données, elles permettent de décrire *ce qui doit* être calculé plutôt que *comment cela doit* être calculé. La *composition descriptionnelle* [7, 8, 23] est une transformation qui élimine les constructions de structures de données intermédiaires lors de la composition de deux grammaires attribuées.

Nous avons étudié [5, 6, 4] les similarités et les différences entre la composition descriptionnelle et un large sous ensemble des méthodes de déforestation *in calculational form* [9, 25, 10]. Nous avons montré que dans les cas les plus simples (fonctions du premier ordre), les deux techniques aboutissent à des résultats équivalents. Mais nous étions également convaincus que, au moins pour une large classe de programmes, la composition descriptionnelle était plus puissante.

Pour illustrer cette conviction, prenons l'exemple de *rev*, la fonction qui inverse une liste d'éléments, et de *flat*, la fonction qui construit une liste à partir d'un arbre binaire, en parcourant ses feuilles de gauche à droite[1] (dans les deux fonctions, les paramètres l sont initialisés avec nil) :

```
let rev x l = case x with
  cons head tail →
    rev tail (cons head l)
  nil → l
```

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

[1] Précisons que cette fonction construit la liste en commencant par *nil*, et que comme elle parcourt l'arbre de gauche à droite, le premier élément de la liste est la feuille la plus à droite de l'arbre.

La composition classique de ces deux fonctions permet d'obtenir la fonction suivante

$$\texttt{let } \mathit{rev\_flat}\ t = \mathit{rev}\ (\mathit{flat}\ t\ \mathit{nil})\ \mathit{nil}$$

dans laquelle la liste construite par *flat* est une structure intermédiaire consumée par *rev*. En traduisant *rev* et *flat* en grammaires attribuées, l'application d'une transformation basée sur la composition descriptionnelle génère une nouvelle grammaire attribuée correspondant à la fonction suivante :

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

Dans cette fonction, le résultat définitif est construit directement, en parcourant les feuilles de l'arbre de droite à gauche, et la liste intermédiaire n'est plus construite. À notre connaissance, aucune méthode de déforestation fonctionnelle ne permet d'aboutir à un tel résultat sur cet exemple.

La contribution principale de cet article est de promouvoir la composition descriptionnelle au rang de *méthode efficace de déforestation de programmes fonctionnels.* Pour ce faire, notre étude comporte différentes étapes :

- la définition d'une traduction de programmes fonctionnels en notation par grammaires attribuées ;
- cette traduction nécessite la définition d'une nouvelle sorte d'évaluation symbolique pour les grammaires attribuées ;
- l'application de la composition descriptionnelle dans ce contexte ;
- finalement, la mise en commun de la composition descriptionnelle avec une élimination de règles de copies et avec de l'évaluation symbolique, pour définir une nouvelle transformation de programmes appelée *composition symbolique.*

Les deux premiers points ci-dessus définissent une traduction de programmes fonctionnels en grammaires attribuées que nous appellerons traduction FP-to-AG. Par ailleurs, la traduction inverse (AG-to-FP) est possible, basée sur des techniques bien connues [12]. Grâce à ces deux traductions complémentaires, la composition descriptionnelle peut être pleinement appliquée dans le cadre des programmes fonctionnels, et vue comme une transformation source à source. Ceci nous permet alors

de caractériser une classe de programmes pour lesquels la composition symbolique déforeste *mieux* que les autres méthodes fonctionnelles.

Cet article est organisé comme suit. La section 2 définit la traduction d'un programme fonctionnel en une grammaire attribuée équivalente. La section 3 présente la composition descriptionnelle ainsi que d'autres transformations permettant de l'appliquer pleinement aux grammaires attribuées générées par la traduction FP-to-AG. Finalement, la section 4 compare nos résultats aux travaux connexes et dresse le bilan des améliorations apportées, tant pour les méthodes de déforestation fonctionnelles que pour les transformations de grammaires attribuées.

2. La traduction FP-to-AG

L'idée intuitive de la traduction d'un programme fonctionnel en sa notation par grammaires attribuées[2] est la suivante. Chaque appel fonctionnel associé à un schéma de récursion (un *pattern*) doit être démonté en un ensemble d'équations orientées, appelées *règles sémantiques*, sur ce schéma de récursion. Les paramètres du programme fonctionnel deviennent des *attributs* explicites attachés aux variables du *pattern*. Chacune de ces *occurrences d'attributs* est définie par une règle sémantique. Les appels récursifs explicites deviennent donc implicites sur la structure de données sous-jacente, et les règles sémantiques définissent un flot de contrôle explicite.

Cette traduction est décomposée en deux étapes : la *transformation préliminaire* et l'*évaluation symbolique du profil*.

2.1. Langages et notations

Afin de présenter clairement et simplement les étapes basiques des transformations, nous restreignons délibérément notre présentation à une sous-classe des programmes fonctionnels (figure 1).

Nous ne traitons donc ici que les programmes à filtrage (*pattern matching*) explicite (de hauteur 1) sans expression *if-then-else*. Notons également que les instructions `case` ne peuvent se trouver qu'en tête des expressions du corps de la fonction. Les *grammaires attribuées dynamiques* [22] sont une façon de prendre en compte les expression *if-then-else* et les expressions `let` locales peuvent être facilement ajoutées, mais nous ne développerons pas ces points ici. Les autres formalismes de déforestation traitent des classes de programmes similaires (cf. système HYLO [19])

[2] Cette notation n'est pas la notation classiquement utilisée.

$$
\begin{array}{lcl}
prog & ::= & \{def\}^* \\
def & ::= & \texttt{let } f\ \overline{x} = exp \\
 & | & \texttt{let } f\ \overline{x} = \texttt{case } x_k \texttt{ with } \{pat \rightarrow exp\}^+ \\
pat & ::= & c\ \overline{x} \\
exp & ::= & Constantes \\
 & | & x \in Variables \\
 & | & \lambda x.exp \\
 & | & f\ \overline{exp}
\end{array}
$$

Figure 1: Langage fonctionnel

La fonction *length* illustre cette syntaxe :

$$
\begin{array}{l}
\texttt{let } length\ x = \texttt{case } x \texttt{ with} \\
\quad cons\ head\ tail \rightarrow (+\ 1\ (length\ tail)) \\
\quad nil \rightarrow 0
\end{array}
$$

Pour rapprocher les grammaires attribuées (voir figure 2) des spécifications fonctionnelles, les définitions de types algébriques seront préférées aux grammaires indépendantes du contexte classiques [1]. Par exemple, les types *list* et *tree* sont définis comme suit :

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

Une production de grammaire est représentée comme un constructeur de type suivi de variables de paramètres (exemple : *cons head tail*), aussi appelé un *pattern*.

De plus, nos transformations prennent en entrée des programmes fonctionnels bien typés. Cela induit des informations concernant les grammaires attribuées générées. Par exemple, la sorte[3] des attributs et leur type sont directement déduits du programme fonctionnel en entrée.

La notion de *profil* d'une grammaire attribuée est également introduite. Un tel profil représente l'appel de la grammaire attribuée. Il permet de spécifier ses arguments et son résultat. Cette notion étend les grammaires attribuées classiques dans lesquelles la spécification d'arguments était interdite.

L'occurrence de l'attribut a sur la variable x d'un *pattern* est notée $x.a$. Si un attribut est attaché au constructeur du *pattern* courant, son occurrence est notée $this.a$ mais, par abus de notation, nous la noterons

[3]Rappelons qu'il existe deux sortes d'attributs : les *synthétisés* calculés de bas en haut dans la structure d'entrée, et les *hérités* calculés de haut en bas.

bloc	::=	`let` $f = \{f\ \overline{x} \rightarrow \overline{semrule}\}\{pat \rightarrow \overline{semrule}\}^*$
semrule	::=	$occ = exp$
occ	::=	$x.a$

exp est le même que dans la figure 1 et *occ* est ajouté à *Variables*.

Figure 2: Notation grammaire attribuée

souvent simplement a. Pour l'exemple de *length*, nous obtenons donc la grammaire attribuée suivante :

```
let length =                          ; Nom de la grammaire attribuée
  length x →                          ; Profil de length
    result = x.length                 ; result est l'unique attribut du profil
                                      ; length est le seul attribut de x
  cons head tail →                    ; Pattern matching sur cons
    length = (+ 1 tail.length)        ; Définition de l'attribut length
  nil →                               ; Pattern matching sur nil
    length = 0                        ; Définition de l'attribut length
```

Dans la suite de cet article, les notations suivantes seront utilisées :

$\stackrel{def}{=}$	:	définition locale dans un algorithme
$\overline{x}$	:	un n-uplet $x_1, \ldots, x_n$
$x.a = exp$	:	une équation orientée ou règle sémantique définissant l'occurrence d'attribut $x.a$
$[x := y]$	:	substitution de x par y
Σ	:	un ensemble de règles sémantiques
Π	:	un *pattern* avec ses règles sémantiques
$\mathcal{C} \vdash A \Rightarrow B$	:	la transformation de A en B dans le contexte local $\mathcal{C}$
$\mathcal{E}[e]$	:	un terme contenant l'expression e.

2.2. Saturation et transformation préliminaire

La transformation FP-to-AG peut accepter en entrée des définitions de fonctions d'ordre supérieur. Le premier travail est alors de saturer toutes les applications partielles. Cette étape permet de mettre en évidence tous les paramètres possibles, en les extrayant au plus haut niveau de la définition lorsqu'ils sont cachés par des λ-termes internes. Si des termes restent non saturés, ils seront ignorés par toute transformation ultérieure.

Le but de la *transformation préliminaire* est de déterminer la forme générale de la future grammaire attribuée. Elle introduit le profil de la grammaire attribuée ainsi que ses règles sémantiques.

$$\frac{\forall i \quad \overset{exp}{\vdash} a_i \Rightarrow b_i \quad ; \quad f \textit{ est un nom de fonction}}{\overset{exp}{\vdash} (f\ \overline{a}) \Rightarrow (f\ \overline{b}).result} \qquad (App)$$

$$\frac{\overset{exp}{\vdash} e \Rightarrow e'}{f, \{x_j\}_{j\neq k}, x_k \overset{pat}{\vdash} \begin{array}{l} c\ \overline{y} \rightarrow e \ \Rightarrow \\ c\ \overline{y} \rightarrow f = e'[x_k := this][x_j := this.x_j{}^f]_{j\neq k} \end{array}} \qquad (Pattern)$$

$$\frac{\begin{array}{l} \forall i \quad f, \{x_j\}_{j\neq k}, x_k \overset{pat}{\vdash} pat_i \Rightarrow \Pi_i \\ \overline{\Pi} \overset{def}{=} \left(\begin{array}{l} f\ \overline{x} \rightarrow \\ \quad result = x_k.f \\ \quad x_k.x_j{}^f = x_j \quad (j\neq k) \end{array} \right) \cup \Pi_i \end{array}}{\overset{let}{\vdash} \texttt{let}\ f\ \overline{x} = \texttt{case}\ x_k\ \texttt{with}\ \overline{pat} \Rightarrow \texttt{let}\ f = \overline{\Pi}} \qquad (Let)$$

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

$\overset{exp}{\vdash} e \Rightarrow e'$ signifie : l'équation e est traduite en l'équation e'.

$env \overset{pat}{\vdash} p \Rightarrow \mathcal{R}$ signifie : dans l'environnement *env*, l'expression associée au pattern p est traduite en l'ensemble des règles sémantiques $\mathcal{R}$.

$\overset{let}{\vdash} \mathcal{D} \Rightarrow \mathcal{B}$ signifie : la définition de fonction $\mathcal{D}$ est traduite en bloc $\mathcal{B}$.

Figure 3: Transformation préliminaire

L'attribut *result* est défini comme un attribut synthétisé de l'argument filtré (règle *Let*)[4]. Les autres arguments sont transformés en des règles sémantiques définissant les attributs hérités (règle *Let*). Chaque appel de fonction $(f\ \overline{a})$ est transformé en une notation pointée $(f\ \overline{b}).result$ (règle *App*). Cette règle fait la différence entre les appels de fonctions et les appels de constructeurs de type. Chaque expression apparaissant dans un *pattern* est transformée en une règle sémantique qui définit l'attribut synthétisé calculant le résultat (règle *App*). Cela implique des renommages (règle *Pattern*), et en particulier celui de l'argument filtré qui devient *this*.

[4]Pour les appels de fonctions sans argument filtré directement visible, la règle *Let'* est appliquée.

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

$\mathcal{P} \vdash \Pi_1 \Rightarrow \Pi_2$ signifie: dans le programme $\mathcal{P}$ le pattern et son ensemble d'équations Π_1 sont transformés en Π_2.

Figure 4: Évaluation symbolique du profil

La transformation préliminaire de la fonction *flat* (section 1) produit :

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

2.3. Évaluation symbolique du profil

Le résultat de la transformation préliminaire n'est pas encore une véritable grammaire attribuée. Chaque définition de fonction dans le programme initial a été traduite en un bloc (cf. la définition des grammaires attribuées section 2.1). Ce bloc contient le profil de la fonction, ainsi que les *patterns* associés. Cependant, les appels récursifs explicites ont seulement été traduits sous la forme $(f\ \overline{a}).result$. Ces expressions doivent maintenant être transformées en des ensembles de règles sémantiques, en démontant les récursions explicites. Les règles sémantiques définissent alors implicitement la récursion. La figure 4 décrit l'algorithme d'évaluation symbolique du profil qui effectue cette transformation[5].

L'évaluation symbolique du profil projette les règles sémantiques du profil f de la grammaire attribuée partout où une expression $(f\ \overline{a}).result$

[5] L'évaluation symbolique du profil est un cas particulier d'une transformation plus générale (cf. section 3.5).

apparaît. Cette transformation doit être appliquée en profondeur d'abord, et la contrainte *Check* assure que la grammaire attribuée résultante est bien formée, c'est à dire :

C_1 pour chaque occurrence d'attribut utilisée $x.a$:
- si a est synthétisé, l'occurrence d'attribut $this.a$ doit être définie pour chaque *pattern* associé au type de x ;
- si a est hérité, chaque *pattern* dans lequel un paramètre x_i a le type de x doit contenir une définition de $x_i.a$.

C_2 une occurrence d'attribut ne peut pas être définie plusieurs fois dans un même *pattern*.

C_3 les expressions de la forme $(x.a).b$ sont interdites.

Une première version de $Check(c, f, \Sigma)$ vérifie que c et f appartiennent au même bloc. Puisqu'un bloc correspond à une définition de fonction bien typée et saturée, cela garantit que C_1 est satisfait :

- chaque occurrence d'attribut hérité est définie, puisqu'elle correspond à un argument de fonction ;
- l'unique occurrence d'attribut synthétisé est définie, puisqu'elle correspond au résultat de l'unique fonction associée au bloc courant.

Deux autres contraintes doivent être ajoutées à $Check(c, f, \Sigma)$:

- Σ doit vérifier C_2 ; ceci peut être nécessaire pour éviter les problèmes dus aux expressions (non linéaires) de la forme $\mathcal{E}[(f\ t\ a)\ (f\ t\ b)]$.
- Σ doit vérifier C_3 ; ceci peut être nécessaire quand deux fonctions sont composées et font apparaître de telles expressions. Dans un premier temps, elles sont interdites, mais elles permettront plus tard (cf. section 3.2) de localiser les compositions pour les déforester.
- c doit être différent de f pour éviter d'appliquer la règle (PSE) infiniment.

Pour les cas où $Check(c, f, \Sigma)$ n'est pas vérifié, l'expression $(f\ \overline{a}).result$ est simplement réécrite sous la forme d'un appel de fonction $f\ \overline{a}$.

En satisfaisant ces contraintes assez fortes, l'application successive de la saturation, de la transformation préliminaire et de l'évaluation symbolique du profil sur un programme fonctionnel permet d'obtenir

une véritable grammaire attribuée. C'est précisément cette suite de transformations que nous appelons la *traduction FP-to-AG*.

Dans l'exemple *flat* précédent, la règle sémantique associée au *pattern* de constructeur *node* est $flat = (flat\ right\ (flat\ left\ l^{flat}).result).result$. L'application[6] de la règle (PSE) sur cette règle sémantique donne :

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

L'application complète de l'évaluation symbolique du profil pour la fonction *flat* aboutit à la grammaire attribuée bien formée suivante :

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

La traduction FP-to-AG peut accepter une classe étendue de fonctions en relâchant le prédicat $Check$. Une seconde version de $Check(c, f, \Sigma)$ accepte donc que c et f viennent de blocs différents. Dans ce cas, on peut obtenir des blocs générés ne correspondant pas immédiatement à de véritables grammaires attribuées. Cependant, des algorithmes simples de clôture transitive, couramment utilisés dans les grammaires attribuées, permettent grâce à des renommages de recréer une véritable grammaire attribuée à partir de plusieurs blocs.

À ce stade, FP-to-AG construit un ensemble de grammaires attribuées bien formées à partir d'un programme fonctionnel. Il est donc possible d'appliquer les méthodes de déforestation des grammaires attribuées.

[6] Les termes soulignés indiquent les termes sur lesquels les règles sont appliquées.

3. Composition symbolique

3.1. Composition descriptionnelle

Considérons la définition de la fonction *rf* qui aplatit un arbre en une liste, pour finalement inverser cette liste :

$$\texttt{let } rf\ t = (reverse\ (flat\ t\ nil))$$

Partant de cette définition de fonction, FP-to-AG génère la grammaire attribuée *rf* qui fait un appel à la grammaire attribuée *reverse*.

```
let rf =
  rf t →
    result = reverse t.rf
    t.l^rf = nil
  node left right →
    rf = right.rf
    right.l^rf = left.rf
    left.l^rf = l^rf
  leaf n →
    rf = cons n l^rf
```

```
let reverse =
  reverse x →
    result = x.reverse
    x.l^reverse = nil
  cons head tail →
    reverse = tail.reverse
    tail.l^reverse = cons head l^reverse
  nil →
    reverse = l^reverse
```

La grammaire attribuée *reverse* inverse la liste qu'elle accepte en entrée. La grammaire attribuée *rf* construit une liste ($t.rf$) à partir d'un arbre d'entrée t donné ; elle appelle alors la grammaire attribuée *reverse* sur cette liste ($t.rf$) qui renvoit le résultat final *result*. Puisque $t.rf$ est consommée par *reverse*, la construction de cette structure (liste) intermédiaire peut être éliminée. C'est précisément le but de la composition descriptionnelle [7, 8].

Considérons donc la composition de deux grammaires attribuée $\mathcal{G}$ (e.g., *reverse*) et $\mathcal{F}$ (e.g., *rf*), où $\mathcal{F}$ contient un appel à $\mathcal{G}$. L'idée de la composition descriptionnelle est de remplacer chaque règle sémantique dans $\mathcal{F}$ qui spécifie la construction d'une structure intermédiaire — avec un constructeur c —, par la projection de l'ensemble des règles sémantiques de $\mathcal{G}$ associées au *pattern* c. Ce pas de projection est décrit dans la figure 5.

Grâce à un renommage approprié des attributs (un attribut a projeté sur un attribut b génère un nouvel attribut a_b) et grâce au fait que la récursion des structures de données est implicite dans les grammaires attribuées, cette transformation peut être appliquée de manière symbolique et globale.

Dans le *pattern leaf n* de l'exemple *rf*, l'expression $cons\ n\ l^{rf}$ est donc remplacée par la projection des règles sémantiques associées au *pattern*

$$\frac{(c\ \overline{y} \rightarrow \Sigma) \in \mathcal{G} \quad \sigma \stackrel{def}{=} [this.b := x.u\_b][y_i.b := x_i.a_i\_b][y_i := x_i.a_i]}{\mathcal{G}, \mathcal{F} \vdash (x.u = c\ \overline{x.a}) \in \mathcal{F} \Rightarrow \sigma(\Sigma)} \quad (DC)$$

$$\frac{\begin{array}{ll} \forall v \in Attr\_S_{\mathcal{G}}(a) & \Sigma_v \stackrel{def}{=} \overline{x_1.a\_v = x_2.b\_v} \\ \forall w \in Attr\_H_{\mathcal{G}}(a) & \Sigma_w \stackrel{def}{=} \overline{x_2.b\_w = x_1.a\_w} \end{array}}{\mathcal{G}, \mathcal{F} \vdash (x_1.a = x_2.b) \in \mathcal{F} \Rightarrow \Sigma_v \cup \Sigma_w} \quad (Copy)$$

$\mathcal{G}, \mathcal{F} \vdash eq \Rightarrow \Sigma$ signifie : dans la composition de $\mathcal{G}$ et $\mathcal{F}$, la règle semantique eq est remplacée par l'ensemble de règles sémantiques Σ.
$Attr\_S_{\mathcal{G}}(a)$ est l'ensemble des attributs synthétisés de $\mathcal{G}$ et du type de a.
$Attr\_H_{\mathcal{G}}(a)$ est l'ensemble des attributs hérités de $\mathcal{G}$ et du type de a.

Figure 5: Un pas de projection de la composition descriptionnelle

cons dans *reverse*. L'application de la règle (DC)[7] correspondante est présentée ci dessous :

$$\left(\begin{array}{l} cons\ head\ tail \rightarrow \\ \quad this.reverse = tail.reverse \\ \quad tail.l^{reverse} = cons\ head\ this.l^{reverse} \end{array}\right) \in reverse$$

$$\begin{array}{ll} \sigma \stackrel{def}{=} & [this.reverse := this.rf\_reverse] \\ & [this.l^{reverse} := this.rf\_l^{reverse}] \\ & [tail.reverse := this.l^{rf}\_reverse] \\ & [tail.l^{reverse} := this.l^{rf}\_l^{reverse}] \\ & [head := n] \end{array}$$

$$reverse, rf \vdash \{this.rf = cons\ n\ this.l^{rf}\} \in rf \Rightarrow \left\{\begin{array}{l} this.rf\_reverse = this.l^{rf}\_reverse \\ this.l^{rf}\_l^{reverse} = cons\ n\ this.rf\_l^{reverse} \end{array}\right\}$$

La grammaire attribuée complète *rf* déforestée par la composition descriptionnelle est donnée ci-dessous. Quatre attributs ont été générés. La liste finale est construite avec $l^{rf}\_l^{reverse}$ et $rf\_l^{reverse}$ en traversant l'arbre de droite à gauche avant d'être propagée[8] dans le sens inverse.

[7] Par simplicité, dans la règle (DC), $x_i.a_i$ représente aussi les termes tels que x_i. C'est le cas dans la substitution $head := n$ de l'exemple.

[8] Nous traitons dans (3.4) de l'élimination de la plupart de ces propagations.

```
let rf =
  rf t →
    result = t.rf_reverse
    t.rf_l^reverse = nil
    t.l^rf_reverse = t.l^rf_l^reverse
  node left right →
    rf_reverse = right.rf_reverse
    right.rf_l^reverse = rf_l^reverse
    right.l^rf_reverse = left.rf_reverse
    left.rf_l^reverse = right.l^rf_l^reverse
    left.l^rf_reverse = l^rf_reverse
    l^rf_l^reverse = left.l^rf_l^reverse
  leaf n →
    rf_reverse = l^rf_reverse
    l^rf_l^reverse = cons n rf_l^reverse
```

Nous faisons ici quelques remarques concernant la composition descriptionnelle :

1. Dans $\mathcal{F}$, un appel du profil g de la grammaire attribuée $\mathcal{G}$ est traité comme un appel à un constructeur de type (dans la règle DC).

2. Chaque occurrence d'attribut définie doit être utilisée une et une seule fois. Plus précisément, les termes dans lesquels est construit le résultat doivent être linéaires. Cette restriction est vérifiée syntaxiquement avant l'application de la composition descriptionnelle et peut fréquemment être relâchée [23].

3. Toute la construction de la structure intermédiaire doit être visible (ne doit pas être cachée dans l'appel d'une autre fonction).

3.2. Application de la composition descriptionnelle

Le moyen de trouver les sites d'application de la composition descriptionnelle n'a pas encore été abordé. Dans la section 2.3, la contrainte C_3 garantit que les expressions de la forme $(x.f).g$ n'apparaissent pas. Cependant, ces expressions qui correspondent à des compositions de fonctions sont justement des sites potentiels d'application de la composition descriptionnelle. La contrainte C_3 va donc être provisoirement relâchée dans le prédicat *Check*.

Voici un exemple illustrant ce cas :

```
let rf2 t = rev (flat t nil) nil
```

En relâchant la contrainte C_3, l'application de l'évaluation symbolique du profil donne :

$$
\begin{array}{l}
\texttt{let } rf2 = \\
\quad rf2\ t \rightarrow \\
\qquad result = \underline{(t.rf2).rev} \\
\qquad t.l^{rf2} = nil \\
\qquad (t.rf2).l^{rev} = nil \\
\quad \ldots
\end{array}
$$

Dans cet exemple, l'application de la composition descriptionnelle nécessite d'identifier le profil de la grammaire attribuée appelée et d'extraire la grammaire attribuée correspondante.

Ceci est effectué par une transformation appelée *extraction de profil*, qui permet de retrouver la contrainte C_3 vérifiée après la transformation et d'appliquer la composition descriptionnelle. Par manque de place, nous ne présentons pas ici cette transformation technique.

3.3. Résultats et comparaisons

La correction de la composition descriptionnelle a été démontrée dans [7, 8]. En dépit des restrictions de notre traduction FP-to-AG, nous obtenons grâce à la composition descriptionnelle des résultats remarquables : les programmes fonctionnels tels que $reverse \circ flatten$ ou $reverse \circ reverse$ sont déforestés tandis qu'ils ne peuvent pas l'être par les déforestations *in calculational form*.

Nous avions déjà montré [5, 6] que pour des programmes simples, en particulier ceux correspondant aux grammaires attribuées S^1 (qui peuvent être calculées avec un seul attribut synthétisé), l'application des méthodes de déforestation classiques aux programmes fonctionnels aboutissaient au même résultat que l'application de la composition descriptionnelle aux grammaires attribuées correspondantes :

$$\text{FP-to-AG} \circ \text{Déforestation Fonctionnelle} \equiv \text{DC} \circ \text{FP-to-AG}$$

Nous venons de montrer que de plus, pour une classe de fonctions à laquelle rf appartient, l'application de la composition descriptionnelle à des fonctions traduites en grammaires attribuées est plus puissante et plus efficace que les déforestations classiques :

$$\text{FP-to-AG} \circ \text{Déforestation Fonctionnelle} < \text{DC} \circ \text{FP-to-AG}$$

3.4. Élimination des règles de copie

Les grammaires attribuées, et particulièrement celles générées par la composition descriptionnelle, peuvent contenir beaucoup de règles de copie inutiles, qui ne font que de propager des valeurs d'attributs ou des constantes le long de la structure d'entrée. Cependant, une analyse statique globale sur la grammaire attribuée permet d'éliminer la plupart de ces règles de copie [23].

Dans l'exemple de *rf*, l'élimination des règles de copie permet d'obtenir la grammaire attribuée suivante :

```
let rf =
  rf t →
    result = t.rf_rev
    t.rf_l^rev = nil
  node left right →
    rf_rev = left.rf_rev
    right.rf_l^rev = rf_l^rev
    left.rf_l^rev = right.rf_rev
  leaf n →
    rf_rev = cons n rf_l^rev
```

Tout comme la fonction initiale *flat*, la grammaire attribuée *rf* générée par FP-to-AG construit son résultat en traversant l'arbre d'entrée de la gauche vers la droite, et ensuite en inversant la liste obtenue (début de la section 3.1). Après l'application de la composition descriptionnelle, la liste finale est directement construite de la droite vers la gauche, puis seulement propagée en sens inverse de sa construction. L'élimination des règles de copies permet dans ce cas d'obtenir la dernière version ci-dessus, où la liste finale est simplement et directement construite de la droite vers la gauche. Finalement, en générant un évaluateur fonctionnel [7, 12] pour cette dernière grammaire attribuée, le programme obtenu est la fonction *rev_flat* présentée dans l'introduction.

3.5. Évaluation symbolique

L'évaluation symbolique du profil peut être généralisée en une nouvelle *évaluation symbolique* qui effectue à la fois l'évaluation symbolique du profil et de l'évaluation partielle sur les termes constants. De plus, cette méthode est aussi appliquée à l'intérieur de la composition descriptionnelle pour projeter les règles sémantiques sur les règles de construction quand ces dernières contiennent plus d'un constructeur[9].

[9]Dans l'article original sur la composition descriptionnelle [7] ce cas était déjà résolu et connu sous le nom de termes imbriqués (*nested-terms*).

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

Figure 6: Evaluation symbolique

L'idée de l'évaluation symbolique est de projeter récursivement les règles sémantiques sur les termes finis et d'éliminer les attributs intermédiaires qui sont définis et utilisés dans l'ensemble de règles sémantiques produit (voir figure 6).

Évaluation partielle

L'application au terme sur $u = (cons\ a\ (cons\ b\ nil)).rev$ d'un pas d'évaluation symbolique donne :

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

Deux pas supplémentaires d'évaluation symbolique permettent d'obtenir :

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

Cette transformation permet donc d'effectuer de l'évaluation partielle sur les termes finis.

4. Travaux connexes et interprétation des résultats

Déforestation classique

Dans la programmation fonctionnelle, la plupart des méthodes de déforestation dirigées par la structure sont basées sur les notions catégorielles comme les foncteurs, les catamorphismes et les hylomorphismes. Ces méthodes s'appuient sur des lois fondamentales telles que le *Théorème de Promotion* [18], et utilisent des opérateurs de contrôle génériques pour capturer à la fois le schéma de récursion des fonctions et celui des types de données. Tout d'abord, la *Shortcut Deforestation* [9] avec les règles d'élimination `foldr/buildr` a rendu ceci possible pour le type liste. Ensuite, l'*Algorithme de Normalisation* [25, 15] a permis de généraliser la déforestation afin qu'elle fonctionne sur tous les types de données, grâce à la génération automatique de *foncteurs* à partir des définitions des types algébriques. Mais ces foncteurs étaient trop isomorphiques aux types. Pour résoudre ce problème, les *hylomorphismes sous forme de triplets* [26] ont été introduits, mais la notion de foncteur est toujours sous-jacente. Les systèmes tels que ADL et HYLO [19, 13] sont basés sur ce formalisme, et permettent de déforester des programmes complexes de manière automatisée.

Amélioration de la déforestation

En dépit de ces raffinements successifs et de ces généralisations, une classe de programmes restait non déforestable (e.g., $reverse \circ reverse$ et $reverse \circ flatten$ appartiennent à cette classe). De notre point de vue, les remarques informelles suivantes permettent de caractériser cette classe.

Les méthodes fonctionnelles utilisent toujours des foncteurs pour diriger les transformations et les calculs, tandis que la composition descriptionnelle et les grammaires attribuées ne les utilisent pas. L'exemple de *reverse* est très significatif :

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

Soit F_{rev} le foncteur qui dirige la récursion de cette fonction. F_{rev} ne suit pas la construction de la liste résultat de *reverse* ; en suivant les appels récursifs de *rev*, la construction du résultat est cachée. Plus précisément, le constructeur $\underline{cons}$ est caché dans le second paramètre de l'appel de $\underline{rev}$. La déforestation ne peut donc pas atteindre ce constructeur.

Dans les grammaires attribuées, la composition descriptionnelle atteint chaque constructeur du résultat, directement à partir de la spécification. Cela ne nécessite aucune notion abstraite supplémentaire au formalisme initial, telle que les foncteurs. Ceci est dû au fait que les constructions du résultat restent visibles dans les spécifications, même si elles ne suivent pas le foncteur de la récursion. Nous pensons que c'est la raison pour laquelle la composition descriptionnelle est capable d'aboutir à une déforestation plus puissante et plus efficace. Par exemple, dans la section 3.1 le *cons* ci-dessus est déforesté.

Pour conclure ici, une importante contribution de cet article est de montrer que la composition descriptionnelle est réellement une méthode de déforestation générale. Grâce à FP-to-AG, la composition descriptionnelle n'est plus restreinte aux grammaires attribuées.

Amélioration pour les grammaires attribuées

Dans le domaine des grammaires attribuées, ces travaux permettent une intégration plus complète et plus utilisable de la composition descriptionnelle. L'idée initiale de Ganzinger et Giegrich était d'appliquer la composition descriptionnelle uniquement sur deux grammaires attribuées séparées. La conjonction de la composition descriptionnelle avec une nouvelle évaluation symbolique lui permet d'être appliquée sur des expressions à l'intérieur même d'une grammaire attribuée. De plus, tous les termes finis sont évalués par ces transformations. Dans ce contexte, l'évaluation partielle devient un cas particulier de la *composition symbolique*.

Pour finir, rappelons que le formalisme des grammaires attribuées n'est pas seulement une notation abstraite permettant d'écrire des équations sémantiques. C'est également en lui même un langage de programmation complet, reconnu pour sa puissance et son efficacité dans la réalisation de larges applications comme les compilateurs. De plus, pour effectuer de la déforestation, le simple formalisme initial des grammaires attribuées n'a requis aucune extension — même dans les situations complexes.

5. Conclusion

Le principal objectif de cet article est de montrer qu'il est possible de transposer (interpréter) les techniques de transformations de grammaires attribuées en termes de transformations de programmes fonctionnels. Ceci nous permet de montrer que, grâce à son algorithme fondamental de composition symbolique, notre déforestation donne de meilleurs résultats

que les techniques développées par la communauté fonctionnelle. Ces résultats renforcent notre conviction de la simplicité et de la puissance du formalisme des grammaires attribuées pour de telles transformations. La transformation FP-to-AG, présentée ici sous sa forme la plus simple, ainsi que la transformation inverse de Johnsson, doivent être considérées comme des outils auxiliaires. Pour une utilisation pratique de cette déforestation, la transformation FP-to-AG pourrait être approfondie et étendue, mais cela ne remettrait pas en cause la puissance intrinsèque de notre composition symbolique. Ce problème n'est d'ailleurs pas spécifique aux grammaires attribuées, pusiqu'il apparaît également dans les systèmes *in calculation form* (cf. HYLO [19]).

Par ailleurs, nous avons étendu le mécanisme de base de la composition descriptionnelle en un mécanisme plus puissant : la composition symbolique. Cette extension permet d'une part de l'utiliser en terme d'évaluation partielle, et d'autre part de généraliser son utilisation. Elle s'applique désormais sur des termes comportant des compositions de fonctions, et non plus simplement sur une composition de deux grammaires attribuées prises isolément de tout contexte [8]. Du point de vue de la communauté des grammaires attribuées, ceci constitue la principale contribution de cet article.

L'étude de ces transformations s'inscrit dans un cadre plus large : celui de la *généricité dans les grammaires attribuées*. Le principe de cette forme de généricité, dont la composition symbolique est l'outil de base, est d'abstraire un programme pour pouvoir le spécialiser dans divers contextes. Des approches semblables à cette forme de réutilisabilité émergent depuis quelques temps dans différents paradigmes de programmation (programmation *polytypique* [11], programmation *adaptive* [21]). Nous commençons à comparer [3] ces approches à nos outils de généricité [16, 17, 24, 23, 2], qui ont été implantés dans notre système FNC-2. Tout comme dans le cadre de la déforestation, il apparaît que les grammaires attribuées, particulièrement adaptées aux transformations de programmes, doivent être vues comme une représentation abstraite d'une spécification plutôt que comme un langage de programmation.

Remerciements Nous sommes reconnaissants à Dick Kieburtz pour nous avoir encouragé après des discussions fructueuses sur ces travaux. Merci également à Françoise Bellegarde, John Boyland, Guy Tremblay et Alberto Pardo pour leurs commentaires et leurs remarques constructives sur les premières versions de cet article.

Bibliographie

[1] Chirica (Laurian M.) et Martin (David F.). – An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, vol. 13, n1, 1979, pp. 1–27. – See also: report TRCS78-2, Dept. of Elec. Eng. and Computer Science, University of California, Santa Barbara, CA (October 1978).

[2] Correnson (Loïc). – *Généricité dans les Grammaires Attribuées.* – Rapport de stage d'option, École Polytechnique, 1996.

[3] Correnson (Loïc). – *Programmation Polytypique avec les Grammaires Attribuées.* – Rapport de DEA, Université de Paris VII, septembre 1997.

[4] Correnson (Loïc), Duris (Etienne), Parigot (Didier) et Roussel (Gilles). – Attribute grammars and functional programming deforestation. *In : Fourth International Static Analysis Symposium – Poster Session.* – Paris, France, septembre 1997.

[5] Duris (Etienne), Parigot (Didier), Roussel (Gilles) et Jourdan (Martin). – Grammaires attribuées et folds : opérateurs de contrôle génériques. *In : Journées Francophones des Langages Applicatifs.* – Dolomieu, France, janvier 1997.

[6] Duris (Etienne), Parigot (Didier), Roussel (Gilles) et Jourdan (Martin). – *Structure-directed Genericity in Functional Programming and Attribute Grammars.* – Rapport de Recherche n3105, INRIA, février 1997.

[7] Ganzinger (Harald) et Giegerich (Robert). – Attribute coupled grammars. *In : ACM SIGPLAN '84 Symp. on Compiler Construction*, pp. 157–170. – Montréal, juin 1984. Published as *ACM SIGPLAN Notices*, 19(6).

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

[9] Gill (Andrew), Launchbury (John) et Jones (Simon L Peyton). – A short cut to deforestation. *In : Conf. on Functional Programming and Computer Architecture (FPCA'93).* pp. 223–232. – Copenhagen, Denmark, juin 1993.

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

[11] Jansson (P.) et Jeuring (J.). – PolyP - a polytypic programming language extension. *In : 24th ACM Symp. on Principles of Programming Languages.*

[12] Johnsson (Thomas). – Attribute grammars as a functional programming paradigm. *In : Func. Prog. Languages and Computer Architecture*, éd. par Kahn (Gilles), pp. 154–173. – New York–Heidelberg–Berlin, Springer-Verlag, septembre 1987. Portland.

[13] Kieburtz (Richard) et Lewis (Jeffrey). – *Algebraic Design Language.* – Rapport technique, Oregon Graduate Institute, 1994.

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

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

[16] Le Bellec (Carole). – *La généricité et les grammaires attribuées*. – Thèse de PhD, Département de Mathématiques et d'Informatique, Université d'Orléans, 1993.

[17] Le Bellec (Carole), Jourdan (Martin), Parigot (Didier) et Roussel (Gilles). – Specification and Implementation of Grammar Coupling Using Attribute Grammars. *In : Programming Language Implementation and Logic Programming (PLILP '93)*, éd. par Bruynooghe (Maurice) et Penjam (Jaan). pp. 123–136. – Tallinn, août 1993.

[18] Meijer (E.), Fokkinga (M. M.) et Paterson (R.). – Functional programming with bananas, lenses, envelopes and barbed wire. *In : Conf. on Functional Programming and Computer Architecture (FPCA'91)*. pp. 124–144. – Cambridge, septembre 1991.

[19] Onoue (Y.), Hu (Z.), Iwasaki (H.) et Takeichi (M.). – A calculational fusion system HYLO. *In : In Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*. – Le Bischenberg, France, février 1997.

[20] Paakki (Jukka). – Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, vol. 27, n 2, juin 1995, pp. 196–255.

[21] Palsberg (Jens), Patt-Shamir (Boaz) et Lieberherr (Karl). – A new approach to compiling adaptive programs. *In : European Symposium on Programming*, éd. par Nielson (Hanne Riis). pp. 280–295. – Linkoping, Sweden, 1996.

[22] Parigot (Didier), Roussel (Gilles), Jourdan (Martin) et Duris (Etienne). – Dynamic Attribute Grammars. *In : Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, éd. par Kuchen (Herbert) et Swierstra (S. Doaitse). pp. 122–136. – Aachen, septembre 1996.

[23] Roussel (Gilles). – *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées*. – Thèse de PhD, Département d'Informatique, Université de Paris 6, mars 1994.

[24] Roussel (Gilles), Parigot (Didier) et Jourdan (Martin). – Coupling Evaluators for Attribute Coupled Grammars. *In : 5th Int. Conf. on Compiler Construction (CC' 94)*, éd. par Fritzson (Peter A.). pp. 52–67. – Edinburgh, avril 1994.

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

[26] Takano (Akihiko) et Meijer (Erik). – Shortcut deforestation in calculational form. *In : Conf. on Func. Prog. Languages and Computer Architecture*. pp. 306–313. – La Jolla, CA, USA, 1995.

[27] Wadler (Philip). – Deforestation: Transforming Programs to Eliminate Trees. *In : European Symposium on Programming (ESOP '88)*, éd. par Ganzinger (Harald). pp. 344–358. – Nancy, mars 1988.