Grammaires attribuées et folds : opérateurs de contrôle génériques

Etienne Duris & Didier Parigot
& Gilles Roussel[1] & Martin Jourdan

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

Résumé

Les opérateurs de contrôle génériques tels que *fold* ont été introduits en programmation fonctionnelle pour augmenter la puissance et le champ d'application des transformations fondées sur la structure des données. Ceci est possible en rendant cette structure plus explicite dans la spécification des programmes.

Nous considérons que cette caractéristique fondamentale est l'un des concepts de base des grammaires attribuées. Dans cet article, nous exposons informellement les similitudes qui existent entre le formalisme du *fold* et la spécification par grammaires attribuées. Nous comparons également leurs méthodes respectives d'élimination des structures intermédiaires introduites lors de la composition de fonctions (notion de déforestation ou de fusion) : l'algorithme de normalisation pour les programmes exprimés à l'aide de *folds* et la composition descriptionnelle pour les grammaires attribuées.

Le but principal de cet article est de présenter intuitivement chacun de ces deux paradigmes, ainsi que leurs similitudes qui offrent des possibilités de fertilisation croisée.

1. Introduction

Ces dernières années, la communauté de la programmation fonctionnelle a étudié différentes techniques d'optimisation de programme, en particulier l'élimination des structures intermédiaires "inutiles" apparaissant lors de la composition de fonctions. La première approche était fondée sur les techniques d'évaluation partielle. Ensuite, une approche symbolique a été proposée par Wadler : la *déforestation* [27], fondée sur les transformations de "pliage-dépliage" (ou "fold and unfold") introduites dans [3]. Plus récemment, différents formalismes ont été proposés pour étendre la puissance de ces transformations de programmes fonctionnels [17, 9, 13, 23, 10, 16]. Nous nous sommes particulièrement intéressés à l'opérateur de contrôle générique *fold*, tel qu'il est présenté par Sheard et Fegaras [23], même si ce formalisme a, depuis, été étendu ou généralisé [10, 16, 24]. Elles sont fondées sur un style de programmation *dirigé par la structure*, et l'opérateur *fold* permet, à l'aide de l'*algorithme de normalisation*, d'effectuer de la *fusion* ou de la *déforestation* de programmes.

Par ailleurs, ce problème de l'élimination des structures intermédiaires qui apparaissent lors de la composition de fonctions a été étudié dans le contexte d'un autre paradigme bien connu de programmation dirigé par la structure (ou syntaxe), à savoir les Grammaires Attribuées (GAs). Elles ont été introduites il y a une trentaine d'années par Knuth [15] et ont été largement étudiées depuis [1, 18, 19]. Une GA est une spécification déclarative qui décrit comment des attributs sont calculés par des règles pour une grammaire particulière (programmation dirigée par la syntaxe). Les GAs ont été reconnues comme ayant deux qualités importantes : d'une part, elles possèdent une *décomposition structurelle* naturelle ; d'autre part, elles sont *déclaratives* en ce sens que le programmeur spécifie uniquement les règles permettant de calculer les attributs, mais pas l'ordre dans lequel elles doivent être appliquées.

Dans ce contexte, le problème de l'élimination des structures intermédiaires a été étudié par Ganzinger et Giegerich, et résolu par leur algorithme de *composition descriptionnelle* [12, 2, 21, 22]. Étant données deux GAs telles que le résultat de la première est l'argument de la seconde, la composition descriptionnelle permet de produire une nouvelle GA qui a la même sémantique que la composition des GAs originales, mais qui ne construit plus la structure intermédiaire.

Les opérateurs de contrôle génériques tels que *fold* permettent de décrire des modèles de récursion de manière uniforme pour une large classe de types et, en ce sens, ils ressemblent beaucoup aux spécifications par GA. De plus, l'algorithme de normalisation semble très proche de la composition descriptionnelle ; en plus d'avoir le même objectif, ces deux techniques possèdent le même caractère de *localité* par rapport à la structure.

Nous allons donc présenter des comparaisons entre l'opérateur de contrôle générique *fold* et les GAs en tant que paradigmes de programmation dirigés par la structure, en nous intéressant à la fois à leur puissance d'expression et aux possibilités qu'ils offrent pour l'élimination des structures intermédiaires.

La suite de cet article (qui est une version allégée de [6]) est divisée en deux sections. La première présente une comparaison entre les *folds* de premier ordre et les GAs. En considérant qu'une GA est définie sur un type algébrique (vu comme la grammaire sous-jacente), nous montrons que, du point de vue de la spécification, les programmes exprimés par des *folds* de premier ordre sur un type algébrique donné peuvent facilement être traduits en GAs purement synthétisées. Comme la sémantique (calcul du résultat) d'une fonction écrite à l'aide d'un *fold* est fondée sur la notion de *foncteurs* [23], ces derniers peuvent être considérés comme des évaluateurs particuliers pour les GAs purement synthétisées à un seul attribut [11]. Nous présentons ensuite informellement les mécanismes de déforestation de chaque formalisme, et montrons que l'algorithme de normalisation pour les *folds* du premier ordre et la composition descriptionnelle pour les GAs purement synthétisées à un seul attribut sont équivalents (ils produisent les mêmes résultats).

La seconde section présente deux approches permettant des transformations de programmes plus puissantes ; tout d'abord justifiées intuitivement, nous décrivons l'approche avec des *folds* de second ordre et celle avec des attributs hérités, et nous détaillons leurs différences essentielles et leur méthodes de déforestation respectives. Pour terminer, nous proposons différentes extensions de ce travail.

2. Folds du premier ordre et grammaires attribuées purement synthétisées

Les opérateurs de contrôle génériques *fold* sont définis sur des ensembles de *types mutuellement récursifs* [23]. Un tel type algébrique est défini par un ensemble de fonctions de construction (constructeurs) manipulant des variables de type. La figure 1 présente quelques exemples de définitions de types algébriques simples. Les GAs, qui sont traditionnellement spécifiées sur des grammaires non contextuelles (CFG), peuvent être définies sur des types algébriques comme ceux utilisés pour les *folds*, en considérant les constructeurs comme les productions de la grammaire sous-jacente, et en donnant une notion de *profil* pour une GA (signature de la fonction représentée par la GA). Par exemple, un constructeur de type algébrique $C_i(t_{i,1}, \ldots, t_{i,m_i})$ peut être considéré comme la production $C_i \rightarrow t_{i,1}, \ldots, t_{i,m_i}$ d'une grammaire définissant ce type, où tous les symboles $t_{i,j}$ sont des variables de type vues comme des occurrences de terminaux ou de non-terminaux. Ainsi, les productions correspondant aux types algébriques de la figure 1 sont présentées dans la figure 2.

list(α)	=	Nil | Cons(α,list(α))
tree(α)	=	Tip(α) | Node(tree(α),tree(α))
int	=	Zero | Succ(int)

Figure 1: Exemples de définitions de types algébriques

Type list*:*	*Type* tree*:*	*Type* int*:*
Cons $\rightarrow$ α list	Node $\rightarrow$ tree tree	Succ $\rightarrow$ int
Nil $\rightarrow$	Tip $\rightarrow$ α	Zero $\rightarrow$

Figure 2: Productions correspondant aux types de la Fig. 1

2.1. Les notations : Fold *vs.* GA

A l'aide de l'opérateur de contrôle générique *fold*, il est possible de décrire des programmes. Sur le type *list*, les fonctions de longueur d'une liste (figure 3) et de concaténation de deux listes (figure 5) sont assez simples à exprimer [23].

Les deux lambda-expressions de la figure 3 sont appelées *fonctions accumulatives de résultat* et représentent les calculs à effectuer sur chacun des constructeurs du type *list* : l'une, pour le constructeur *Nil*, n'a aucun paramètre ; l'autre, pour le constructeur *Cons*, accepte deux paramètres, déterminés par la définition générique du fold^{list} qui sera donnée dans la définition 2.1.

Pour un type donné, une GA est constituée d'un ensemble d'attributs, d'un ensemble de *règles sémantiques* permettant de calculer ces attributs pour chaque production (constructeur) et d'un *profil.* On distingue deux

$$\begin{array}{ll} \text{length(x)} = \text{fold}^{list}\ (& \lambda().\text{Zero}, \\ & \lambda(\text{a,r}).\text{Succ(r)}\ )\ \text{x} \end{array}$$

Figure 3: Définition de *length* avec un *fold*

`Nil ->`

$$f_{Nil,s_{\uparrow_{Nil}}} \quad : \mathrm{s}_{\uparrow_{Nil}} = \text{Zero} \qquad \text{i.e. } (\lambda().\text{Zero})()$$

`Cons -> a list`

$$f_{Cons,s_{\uparrow_{Cons}}} \quad : \mathrm{s}_{\uparrow_{Cons}} = \text{Succ}(\mathrm{s}_{\uparrow_{list}}) \quad \text{i.e. } (\lambda(\text{a,r}).\text{Succ(r)})(\text{a},\mathrm{s}_{\uparrow_{list}})$$

Figure 4: Définition de *length* en GA

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

Figure 5: Définition de *append* avec un *fold*

```
Nil ->
```
$f_{Nil,v_{\uparrow_{Nil}}}$: $\text{v}_{\uparrow_{Nil}} = (\lambda().\text{y})()$
```
Cons -> a list
```
$f_{Cons,v_{\uparrow_{Cons}}}$: $\text{v}_{\uparrow_{Cons}} = (\lambda(\text{a,r}).\text{Cons(a,r)})(\text{a,v}_{\uparrow_{list}})$

Figure 6: Définition de *append* en GA

types d'attributs : les attributs synthétisés (notés avec $_\uparrow$) qui permettent de remonter des calculs dans la structure de données et les attributs hérités (notés avec $_\downarrow$) qui permettent de descendre des calculs dans la structure de données.

Sur les productions du type *list*, il est alors possible de décrire la fonction *length* avec une GA (figure 4). Les règles sémantiques $f_{Nil,s_{\uparrow_{Nil}}}$ et $f_{Cons,s_{\uparrow_{Cons}}}$ pour les productions (constructeurs) *Nil* et *Cons* dans la figure 4 sont aisément représentables sous la forme de lambda-expressions, et correspondent aux fonctions accumulatives de résultat du programme en *fold* représentant la fonction *length* (figure 3). Plus généralement, nous notons par exemple $f_{C,a_{\downarrow_x}}$ la règle sémantique permettant de définir l'occurrence de l'attribut hérité $a_\downarrow$ sur le non-terminal x pour la production C.

La différence principale entre ces deux syntaxes (*fold* et GA) est que chacune des variables utilisées dans une GA (occurrence d'attribut) est explicitement mentionnée par un nom spécifique. Par exemple, dans la construction `Cons ->  a list` de la figure 4, le résultat attendu sur `list` est explicitement nommé $s_{\uparrow_{list}}$ tandis que, sous la forme de *fold*, ce résultat est implicitement représenté par r (appelé *variable accumulative de résultat*). Cette notation explicite des GAs permet d'exprimer des fonctions qui retournent plus d'un résultat (plusieurs attributs synthétisés), et qui possèdent plus d'un paramètre (plusieurs attributs hérités). Dans cette section, nous nous contentons de comparer les GAs purement synthétisées (qui n'utilisent qu'un attribut synthétisé) avec les *folds* du premier ordre. Cependant, nous pouvons déjà remarquer que les règles sémantiques de la forme en GA sont exactement les fonctions accumulatives de la forme en *fold*, appliquées aux paramètres explicites (les occurrences d'attributs appropriées).

Pour l'exemple de *length* (figure 4), le profil de la GA correspondante est $length(root : list) \rightarrow s_{\uparrow} : int$, ce qui signifie que $root$ représente la liste d'entrée et que le résultat de la GA, obtenu dans l'attribut synthétisé $s_{\uparrow}$, est de type int. La fonction *append(x,y)* de concaténation de deux listes (figure 5), peut également être spécifiée par une GA purement synthétisée (figure 6). Le profil de cette GA est $append(root : list, y : list) \rightarrow v_{\uparrow} : list$, où le premier paramètre $root$ est la structure de donnée, et le second y est une variable ; le résultat de type $list$ est calculé dans l'attribut $v_{\uparrow}$.

2.2. Foncteurs et évaluateurs d'attributs

Après les notations et la puissance d'expression, nous allons nous intéresser à la sémantique et à l'évaluation des *folds* et des GAs. La sémantique d'une fonction exprimée avec un *fold* est donnée par la définition de cet opérateur, qui utilise la notion de *foncteur* [23]. Sans rentrer dans le détail du calcul de ces foncteurs, qui sont déterminés de façon statique à partir des constructeurs d'un type algébrique, nous donnons ci-dessous les équations définissant l'opérateur *fold* pour le type simple *list*.

Définition 2.1 (Opérateur *fold* pour le type *list*) *L'opérateur* fold^{list} *est défini sur chaque constructeur du type* list *par:*

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

Cette définition permet d'évaluer la fonction *length* de la figure 3, puisqu'elle permet d'identifier les paramètres de f_c (c'est à dire a et r). Le rôle de l'opérateur *fold*^{list} est de déterminer les paramètres à fournir aux fonctions accumulatives. Pour le constructeur $Cons(a, l)$, la variable accumulative de résultat est définie récursivement sur l (le reste de la liste). Cet opérateur, fold^{list}, est défini avec des foncteurs qui sont statiquement déterminés à partir du type *list*. Ces foncteurs fournissent donc un sens à l'évaluation des fonctions exprimées avec un *fold*.

Pour le type *list*, chaque calcul est effectué en appliquant récursivement f_c sur le premier élément de la liste et sur le résultat (à venir) du calcul sur le reste de la liste, jusqu'à ce que f_n puisse être appliquée sur le constructeur Nil. Ceci est vrai quelle que soit la sémantique de f_c et f_n : le *fold* est un opérateur de contrôle *générique*, qui pour un type donné est défini une fois pour toutes.

Pour une GA, la sémantique est donnée par la solution du système d'équations associé à l'ensemble des règles sémantiques et des instances d'attributs pour une structure d'entrée particulière, et ceci quelle que soit la méthode utilisée pour résoudre ce système d'équations.

En d'autres termes, pour une spécification (GA) donnée, différentes techniques [8] peuvent être employées pour générer un évaluateur d'attributs, utilisant éventuellement différentes méthodes d'évaluation, mais conduisant toujours à la même solution. Ainsi, la sémantique d'une GA repose uniquement sur sa spécification (la signature de ses règles sémantiques) et elle est indépendante de la méthode d'évaluation.

Dans le cas particulier où une GA représente un programme exprimé en *fold*, les foncteurs spécifient un évaluateur d'attributs qui est correct pour la GA. La classe des GAs correspondant à des programmes exprimés en *fold* du premier ordre est la classe la plus simple des GAs, appelée classe des GAs purement synthétisées, et notée S^1— chaque non-terminal porte un seul attribut synthétisé. Notons qu'il existe des classes plus évoluées de GAs (avec attributs hérités) et de *catamorphismes* (avec des arguments supplémentaires) ; voir [11] pour une traduction des unes vers les autres.

2.3. L'algorithme de normalisation et la composition descriptionnelle

En restant dans le cadre des *folds* du premier ordre et des GAs purement synthétisées, nous allons maintenant comparer leur méthodes de déforestation : l'Algorithme de Normalisation (AN) pour les *folds* de premier ordre et la Composition Descriptionnelle (CD) pour les GAs S^1. Nous allons présenter rapidement chacune de ces méthodes et leurs effets sur l'exemple simple de la composition *length(append(x,y))*. Nous montrerons que les résultats sont similaires, même si les moyens (algorithmes) sont différents.

L'algorithme de normalisation

L'algorithme de normalisation est composé de trois parties :

- la *Généralisation* qui consiste à associer une variable à un terme, et à remplacer ensuite ce terme par la variable associée à chaque fois qu'il est rencontré durant les étapes de l'AN ;
- l'*Application à une Construction* qui est l'application de la définition de l'opérateur *fold* à un constructeur et à ses paramètres (définition 2.1) ; la figure 7 présente un exemple de ce pas de l'AN sur la fonction *length* ;
- la *Promotion* qui est l'étape fondamentale de l'AN. Elle est basée sur le Théorème de Promotion du Fold [23]. Ce théorème établit que la composition d'une fonction g avec une fonction exprimée en

Puisque	length(x) = fold^{list}(λ ().Zero,λ (a,r).Succ(r)) x
et	fold^{list} (f_n, f_c) (Cons(a,l)) = f_c(a,fold^{list}(f_n, f_c) l)
Alors	length(Cons(a,l)) = Succ(length(l))

Figure 7: *Application à une Construction* sur *length*

fold est une nouvelle fonction exprimée en *fold*, dont les fonctions accumulatives dépendent à la fois de la fonction g et des fonctions accumulatives du *fold* initial. Par exemple, pour le type *list*, ce théorème s'écrit :

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

Le Théorème de Promotion assure la validité du *fold* résultant pour des fonctions ϕ construites localement sur chaque constructeur, c'est à dire que chaque fonction accumulative ϕ_i du résultat ne dépend que de la fonction g et de la fonction accumulative f_i du *fold* originel. Nous avions remarqué [5] que l'algorithme de déforestation de Wadler [27] était une transformation plus *globale* que la composition descriptionnelle. Nous allons voir que la CD est une transformation qui possède le même caractère de *localité* que le Théorème de Promotion du Fold.

Dans l'exemple de *length(append)*, le rôle du Théorème de Promotion est de donner une définition pour les fonctions accumulatives ϕ_i et de prouver la correction du *fold* résultant. Il crée donc de nouvelles fonctions accumulatives sur lesquelles il sera possible d'appliquer les pas d'*Application à une Construction* et de *Généralisation*, et ce sont ces deux derniers qui effectuent la véritable déforestation.

Sur l'exemple *length(append)* [23], nous allons dérouler chaque pas de l'AN. Tout d'abord, rappelons les définitions en *fold* de chacune de ces fonctions :

length(x) = fold^{list}(λ().Zero,λ(a,r).Succ(r)) x
append(x,y) = fold^{list}(f_n, f_c) x
avec f_n = λ().y
f_c = λ(a,r).Cons(a,r)

En considérant $g = length$, le Théorème de Promotion du Fold donne :

$$
\begin{array}{lll}
 & \phi_n() & = \text{length}(f_n()) \\
 & & = \text{length(y)} \\
\textit{et} & & \\
 & \phi_c(r_1, r_2) & = \text{length}(f_c(x_1, x_2)) \\
 & & = \text{length}(\text{Cons}(x_1, x_2)) \\
 & & \qquad\qquad \textit{avec } [x_1/r_1,\ \text{length}(x_2)/r_2]
\end{array}
$$

L'*Application à une Construction* donne (d'après la figure 7) :

$$\phi_c(r_1, r_2) = \text{Succ}(\text{length}(x_2))$$

Et pour terminer, la *Généralisation* de length(x_2) par r_2 permet d'obtenir :

$$\phi_c(r_1, r_2) = \text{Succ}(r_2)$$

Ainsi, le résultat de l'AN sur *length(append(x,y))* est le *fold* :

$$
\begin{array}{ll}
\text{fold}^{list}(& \lambda().\text{fold}^{list}(\lambda().\text{Zero},\lambda(\text{a},\text{r}).\text{Succ}(\text{r}))\ \text{y}, \\
 & \lambda(\text{r}_1,\text{r}_2).\text{Succ}(\text{r}_2))\ \text{x}
\end{array}
$$

dans lequel plus aucune structure intermédiaire n'est construite.

La composition descriptionnelle

Considérons les types G_Ω, G_Δ et G_Θ. Pour deux GAs données, $\Omega(G_\Omega) \to G_\Delta$ et $\Delta(G_\Delta) \to G_\Theta$, le but de la CD est de construire une nouvelle GA $(\Delta \circ \Omega)(G_\Omega) \to G_\Theta$ qui possède la même sémantique que l'application successive de Ω et de Δ, de telle sorte que cette nouvelle GA ne construise pas le résultat intermédiaire de type G_Δ. Nous donnerons seulement ici l'idée sur laquelle la CD est basée : la notion de *projection de règles sémantiques* (le lecteur trouvera plus de détails dans [6], et l'algorithme complet dans [12]). Informellement, $(\Delta \circ \Omega)$ est construite à partir de Ω en remplaçant chaque règle sémantique qui construit un terme de G_Δ par la projection des règles sémantiques de Δ sur ce terme. Plus précisément, si une règle sémantique donnée f sur un constructeur $C_i^{G_\Omega}$ construit un terme t de type $C_j^{G_\Delta}$, alors cette règle sémantique f est remplacée par la projection sur le constructeur $C_i^{G_\Omega}$ des règles sémantiques dans Δ du constructeur $C_j^{G_\Delta}$. La CD est une transformation purement syntaxique et ne tient pas compte de la sémantique des règles projetées.[1]

Pour que la GA résultante de la CD soit correcte et pour exprimer les nouvelles règles sémantiques, de nouveaux attributs sont créés lors

[1] En fait, la définition classique de la CD traite de manière spécifique les expressions conditionnelles (*if–then–else*), mais nous ne nous intéressons pas à cette particularité ici.

```
Nil ->
```
$$f_{Nil,vs_{\uparrow_{Nil}}} \ : \mathrm{vs}_{\uparrow_{Nil}} = (\lambda().\mathrm{length(y)})\ ()$$
```
Cons -> a list
```
$$f_{Cons,vs_{\uparrow_{Cons}}} \ : \mathrm{vs}_{\uparrow_{Cons}} = (\lambda(\mathrm{a,r}).\mathrm{Succ(r)})\ (\mathrm{a,vs}_{\uparrow_{list}})$$

Figure 8: La CD de *length* avec *append*

de la projection des règles sémantiques. Les noms de ces attributs sont composés d'un nom d'attribut de Ω concaténé avec le nom d'un attribut de Δ. Pour illustrer un peu mieux cette idée, nous présentons le résultat de la CD sur l'exemple *length(append)* dans la figure 8.

Pour l'exemple de *length* $\circ$ *append*, considérons les définitions en GA de chacune des fonctions (figures 4 et 6). La règle sémantique $f_{Cons,v_{\uparrow_{Cons}}}$ de *append* crée un *Cons*. Donc, la règle sémantique $f_{Cons,s_{\uparrow_{Cons}}}$ de *length* pour la production *Cons* est projetée sur le constructeur *Cons* de *append* conformément à la règle sémantique $f_{Cons,v_{\uparrow_{Cons}}}$ de *append*. Un nouvel attribut vs est créé. La règle sémantique résultante sur la production *Cons* de ($length \circ append$) est $f_{Cons,vs_{\uparrow_{Cons}}}$ (figure 8). Tout comme la différence entre l'évaluation des *folds* et l'évaluation des GAs (la définition des foncteurs dépend uniquement du type tandis que le générateur d'évaluateur dépend de la forme des règles sémantiques), la différence entre l'AN et la CD tient à la connaissance de la méthode d'évaluation. La CD ne nécessite pas du tout cette connaissance. En fait, la méthode d'évaluation choisie pour la GA résultante d'une CD peut être différente des méthodes d'évaluation des GAs d'entrée. On distingue différentes classes de GAs en fonction de leurs techniques d'évaluation possibles. La plus importante de ces classes est celle des GAs *non circulaires*, et elle est stable par la CD, c-à-d que le résultat de la CD de deux GAs non circulaires est encore une GA non circulaire [12].

Pour interpréter la méthode de la CD dans les termes de l'AN, la projection des règles sémantiques de la CD produit directement la version déforestée des ϕ_i, tandis que les ϕ_i obtenues par le Théorème de Promotion doivent encore être déforestées par les étapes d'*Application à une Construction* et de *Généralisation* de l'AN. Il est donc possible de voir le théorème de correction de la CD [12] comme une sorte de Théorème de Promotion plus symbolique, au sens où la preuve de la correction de la CD est indépendante[2] de la méthode d'évaluation des attributs alors que le Théorème de Promotion est fortement lié à la définition des foncteurs. Nous affirmons donc que la CD est une

[2]En prenant garde cependant au problème de la clôture des classes de GAs par la CD.

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

Figure 9: Définition de *reverse* avec un *fold*

Nil ->
$f_{Nil,s_{\uparrow Nil}}$: $\text{s}_{\uparrow Nil} = (\lambda().\text{Nil})\ ()$
Cons -> a list
$f_{Cons,s_{\uparrow Cons}}$: $\text{s}_{\uparrow Cons} = (\lambda(\text{a,r}).\text{append(r,Cons(a,Nil))})\ (\text{a,s}_{\uparrow list})$

Figure 10: La GA correspondant au *fold* pour *reverse*

transformation de source à source, indépendante de toute méthode d'évaluation : c'est une *composition symbolique*, sans aucune étape d'*Application à une Construction*.

Ce pas d'*Application à une Construction* dans l'AN est bien sûr lié à la méthode d'évaluation, c'est-à-dire aux foncteurs. Nous voyons ce pas comme une sorte d'évaluation partielle (*spécialisation*), qui est généralisée par le pas de *Généralisation* de l'AN. Cependant, si l'*Application à une Construction* n'est jamais effectuée sur un lambda-terme lié (terme constant qui n'est pas la spécialisation d'un terme d'entrée), alors l'AN des *folds* du premier ordre est équivalent à la CD des GAs S^1 correspondantes.

3. Folds de second ordre et grammaires attribuées générales

Dans cette partie, nous nous intéressons aux programmes plus complexes, dont la normalisation nécessite une transformation en *folds* de second ordre[3], et qui peuvent être exprimés par des grammaires attribuées *générales* (avec des attributs hérités).

L'Algorithme de Normalisation tel que nous l'avons présenté ne peut pas être appliqué sur tous les programmes écrits à l'aide de *folds*, de même que la Composition Descriptionnelle ne s'applique pas sur toutes les GAs. Par exemple la fonction *reverse* exprimée avec un *fold* du premier ordre (figure 9) n'est pas *potentiellement normalisable* [23]. Pour une raison similaire, la CD ne peut pas être appliquée sur la GA correspondante (figure 10). Pour résoudre ce problème, Sheard et

[3]Un *fold* est de second ordre lorsque l'une de ses fonctions accumulatives de résultat l'est.

Avec le profil $reverse(root : list, h_\downarrow : list) \rightarrow s_\uparrow : list$,
et si $Id = \lambda(x).x$ est la fonction *identité*:

```
Nil ->
```

$f_{Nil,s_{\uparrow_{Nil}}}$: $\mathrm{s}_{\uparrow_{Nil}} = Id\ (\mathrm{h}_{\downarrow_{Nil}})$

```
Cons -> a list
```

$f_{Cons,s_{\uparrow_{Cons}}}$: $\mathrm{s}_{\uparrow_{Cons}} = Id\ (\mathrm{s}_{\uparrow_{list}})$

$f_{Cons,h_{\downarrow_{list}}}$: $\mathrm{h}_{\downarrow_{list}} = (\lambda(\mathrm{a,h}).\mathrm{Cons(a,h)})\ (\mathrm{a,h}_{\downarrow_{Cons}})$

Figure 11: La GA naturelle pour *reverse*

Fegaras [23] ont introduit le *Théorème de Promotion du second ordre.* Cependant, la forme "naturelle" en GA pour exprimer l'inversion de liste n'est pas celle de la figure 10, mais plutôt une GA (figure 11) utilisant un attribut hérité, $h_\downarrow$, initialisé à Nil. On peut alors appliquer directement la CD sur (la composition de) cette GA naturelle (avec elle-même).

Nous allons donc comparer le formalisme utilisant le second-ordre (*fold*) et celui utilisant les attributs hérités (GA), et montrer qu'en dépit de leurs différences apparentes, ils possèdent certaines similitudes intéressantes. Par exemple, la GA purement synthétisée d'ordre supérieur[4] dérivée, grâce à la transformation de Knuth [15, 4], à partir d'une GA donnée (avec des attributs hérités) est semblable au programme *fold* de second ordre correspondant (figure 12).

3.1. L'approche "ordre supérieur"

La fonction *reverse* exprimée en *fold* du premier ordre (figure 9) n'est pas directement normalisable parce que la fonction *append* qu'elle contient agit sur la variable accumulative de résultat r qui est en cours de construction. Plus précisément (voir [23]), le *fold* interne (de la fonction *append*) porte sur r qui est une variable du type *list* liée par le *fold* externe.

Ce problème est résolu dans [23] par l'introduction du Théorème de Promotion de second ordre, qui permet de normaliser un *fold* de second ordre (figure 12) obtenu à partir d'un *fold* de premier ordre (figure 9) grâce à la transformation $\mathcal{F}_\mathcal{G}$.

Cependant, cette transformation $\mathcal{F}_\mathcal{G}$ impose quelques restrictions sur le type de base et sur les fonctions accumulatives de la forme en premier ordre. Le type doit posséder un *zéro-constructeur* (constructeur sans

[4]Dans cet article, "GA d'ordre supérieur" ne désigne pas l'extension du formalisme des GAs de Vogt *et al.* [26], mais une GA (classique) avec des attributs et des règles sémantiques d'ordre supérieur.

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

Figure 12: La forme en *fold* de second ordre pour *reverse*

```
Nil ->
    f_{Nil,s↑Nil}    : s↑Nil = (λ().λ(w).w)
Cons -> a list
    f_{Cons,s↑Cons}  : s↑Cons = (λ(a,s).λ(w).s(Cons(a,w))) (a,s↑list)
```

Figure 13: La GA d'ordre supérieur correspondant à la GA "naturelle" pour *reverse*

arguments, comme *Nil* pour le type *list*), et les fonctions accumulatives apparaissant dans le programme *fold* à transformer doivent utiliser des *fonctions de zéro-remplacement* (comme la fonction *append*). Dans de telles conditions, l'AN permet d'effectuer des déforestations telles que *length(reverse(x))* = *length(x)* ou *reverse(reverse(x))* = *copy(x)*.[5]

Pour appliquer la CD sur des GAs, les constructeurs de la structure intermédiaire à éliminer doivent être directement visibles, pour que la projection des règles sémantiques sur ces constructeurs soit possible et correcte. Dans la GA pour *reverse* (figure 10) correspondant au *fold* en premier ordre, la fonction *append* "cache" les constructeurs et interdit l'application de la CD. Ce problème n'a pas été étudié puisque, dans la forme naturelle en GA, la situation du *append* de notre exemple n'est pas rencontrée.

Cependant, il est intéressant de signaler qu'il existe une transformation classique [15, 4] qui produit, pour toute GA, la GA purement synthétisée d'ordre supérieur correspondante. Par exemple, à partir de la GA naturelle pour *reverse* (figure 11), cette transformation produit la GA S^1 d'ordre supérieur de la figure 13. On peut remarquer la ressemblance entre cette GA S^1 et le *fold* d'ordre supérieur de la figure 12, produit par la transformation $\mathcal{F}_{\mathcal{G}}$ à partir du *fold* initial.

Il paraît intéressant d'étudier la transposition éventuelle de la transformation $\mathcal{F}_{\mathcal{G}}$ au formalisme des GAs. Plus précisément, dans le cas de *reverse*, est il possible de traduire automatiquement la GA de la figure 10 en la GA "naturelle" de la figure 11 ?

[5]En utilisant la transformation $\mathcal{F}_{\mathcal{G}}^{-1}$ définie dans [23].

let (ss,hh) = (reverse ∘ reverse) (x,Nil,hh)
in ss
avec le profil suivant pour (reverse ∘ reverse) :
$(root : list, sh_{\downarrow} : list, hs_{\downarrow} : list) \rightarrow (ss_{\uparrow} : list, hh_{\uparrow} : list)$
et les règles sémantiques suivantes :

```
Nil ->
```

$f_{Nil,hh_{\uparrow_{Nil}}}$: $\mathrm{hh}_{\uparrow_{Nil}} = Id\ (\mathrm{sh}_{\downarrow_{Nil}})$
$f_{Nil,ss_{\uparrow_{Nil}}}$: $\mathrm{ss}_{\uparrow_{Nil}} = Id\ (\mathrm{hs}_{\downarrow_{Nil}})$

```
Cons -> a list
```

$f_{Cons,sh_{\downarrow_{list}}}$: $\mathrm{sh}_{\downarrow_{list}} = Id\ (\mathrm{sh}_{\downarrow_{Cons}})$
$f_{Cons,hh_{\uparrow_{Cons}}}$: $\mathrm{hh}_{\uparrow_{Cons}} = (\lambda(\mathrm{a,hh}).\mathrm{Cons(a,hh)})\ (\mathrm{a,hh}_{\uparrow_{list}})$
$f_{Cons,hs_{\downarrow_{list}}}$: $\mathrm{hs}_{\downarrow_{list}} = Id\ (\mathrm{hs}_{\downarrow_{Cons}})$
$f_{Cons,ss_{\uparrow_{Cons}}}$: $\mathrm{ss}_{\uparrow_{Cons}} = Id\ (\mathrm{ss}_{\uparrow_{list}})$

Figure 14: La GA résultante de la CD de *reverse* avec elle-même

3.2. L'approche "héritée"

Les notions d'attributs synthétisés et hérités sont le concept de base du formalisme des GAs. Introduits dès le début de la théorie des GAs [15], ils permettent de décrire à la fois les calculs "montants" et "descendants" sur la structure. La CD, qui a été définie sur les GAs générales (ayant des attributs synthétisés et hérités), fonctionne indifféremment avec n'importe quelle GA, qu'elle soit S^1 ou non. Par exemple, elle agit directement sur la GA naturelle pour *reverse* (figure 11), qui utilise un attribut hérité.

La figure 14 présente la GA obtenue par l'application de la CD à *reverse* ∘ *reverse*. Dans cet exemple, le profil de la GA résultante joue un rôle important. La fonction représentée par cette GA prend trois arguments : l'argument *root* (la liste) et deux attributs hérités (*sh* et *hs*). Elle retourne deux résultats qui sont les attributs synthétisés *ss* et *hh*. De plus, l'appel de cette GA définit également les dépendances entre les arguments et les résultats : l'argument *hs* dépend (reçoit la valeur) du résultat *hh*.

La GA produite par la CD contient beaucoup de règles de recopie entre les attributs, qui n'ont pas d'autre rôle que de propager les valeurs le long de la structure ; une analyse statique globale simple [21] permet d'éliminer la plupart de ces règles de recopie. À partir de la GA présentée à la figure 14, cette élimination des règles de copie conduit à la GA de recopie d'une liste, ce qui est équivalent au résultat obtenu par l'AN.

3.3. Comparaison et travaux futurs

Dans le cas de la fonction *reverse*, nous avons montré que la CD donnait un résultat équivalent à celui de l'AN, sans recourir à une conversion en ordre supérieur. Pour exprimer la notion d'attribut hérité, le formalisme des *folds* nécessite des fonctions d'ordre supérieur. Cependant, les attributs hérités en GA permettent d'utiliser l'équivalent de variables accumulatives comme arguments de fonctions récursives ; ce pouvoir d'expression n'est pas accessible par les *folds*. Ceci est principalement dû aux définitions des *folds*, qui sont intimement liées aux définitions des foncteurs. Pour introduire la notion d'*hérité* dans le formalisme des *folds*, les foncteurs ne devraient plus être définis uniquement en fonction de la structure du type, mais aussi en fonction des signatures des fonctions accumulatives qui décrivent le résultat et les arguments du calcul ; ceci interdirait la définition statique des foncteurs à partir du type seul. Un tel *foncteur étendu*, qui tiendrait compte à la fois de la structure (type) et de la forme des calculs (signatures des *fonctions accumulatives étendues*), pourrait être défini en utilisant les techniques de génération d'évaluateurs d'attributs.

Une autre différence entre l'AN et la CD tient à la forme du programme d'entrée. En effet, pour les programmes qui ne sont pas directement normalisables au premier ordre, l'AN impose, pour le passage au second ordre, que ce programme soit exprimé avec des fonctions de zéro-remplacement, tandis que la CD ne nécessite pas une telle contrainte. Ces fonctions de zéro-remplacement peuvent être déduites automatiquement à partir du type considéré, si celui-ci possède un zéro-constructeur. Cependant, il existe des types sans zéro-constructeur (le type *tree*, par exemple). Dans [6], nous présentons un exemple où la CD peut être appliquée directement, mais où il semble que l'AN échoue, faute de savoir écrire cet exemple avec une fonction de zéro-remplacement.

Cependant, plus récemment, cette contrainte a été éliminée, en passant systématiquement d'une définition récursive à un *fold* d'ordre supérieur ("Conversion de programmes fonctionnels en programmes algébriques" [10]). Ce travail constitue une amélioration de [23], puisqu'il permet de supprimer la contrainte de représentation avec des fonctions de zéro-remplacement, mais il subsiste des problèmes. Avec l'approche de [10], la version fonctionnelle de l'exemple de *reverse* (version itérative) est convertie en une version inductive, en ordre supérieur, sur laquelle l'AN peut être appliqué tout comme la CD peut être appliquée sur la GA naturelle pour *reverse*. Mais cette conversion, qui utilise un algorithme de normalisation étendu (sous forme de système de règles d'inférences) peut tout de même échouer dans certains cas. De plus, les résultats sont obtenus dans une forme en ordre supérieur, et ne peuvent pas toujours être ramenés au premier ordre ; c'est le cas de l'exemple de *reverse* : la forme naïve avec *append* n'est plus nécessaire, mais la forme naturelle conduit à une version en *fold* du second ordre qu'on ne peut pas ramener au premier ordre.

Par ailleurs, dans [20], nous avons supprimé la nécessité de la présence d'un arbre d'entrée, autrement dit, il est tout à fait possible de considérer le formalisme des GAs comme un langage fonctionnel déclaratif. Dans notre approche, la notion de grammaire dans une GA est plus proche de la notion de schéma de récursion que de programmation dirigée par la syntaxe. Avec nos extensions, la plupart des programmes fonctionnels peuvent être exprimés sous la forme d'une GA équivalente ; il est alors probable que les transformations et les optimisations rendues possibles dans [10] par la programmation algébrique sont aussi accessibles par le formalisme des GAs [25] et ses techniques d'analyse statique. Ce problème n'a pas encore été étudié car, pour des raisons d'efficacité, la plupart des travaux de recherche sur les GAs (sinon tous) évitent d'utiliser le passage à l'ordre supérieur.

4. Conclusion

Même si, à notre connaissance, nos travaux constituent la première tentative de comparaison[6] entre les *folds* et les grammaires attribuées, ces formalismes révèlent déjà des similitudes et des complémentarités.

La première d'entre elles tient à la puissance d'expression des formes d'entrée : un programme en *fold* du premier ordre peut être exprimé avec une grammaire attribuée purement synthétisée à un seul attribut. Dans ce cas, la définition de l'opérateur *fold* du premier ordre, fondée sur les foncteurs, fournit précisément la méthode d'évaluation des attributs pour cette sous-classe de grammaires attribuées. En revanche, les foncteurs sont statiquement déterminés d'après les types algébriques tandis que la plupart des évaluateurs d'attributs (pour les classes de GAs plus générales) sont produits statiquement à la fois à partir des types algébriques et des signatures des règles sémantiques.

Ensuite, même si elles aboutissent à des résultats de déforestation équivalents, l'AN et la CD utilisent des techniques de transformation assez différentes. Plus précisément, la transformation effectuée par l'AN dépend fortement de l'évaluation partielle du terme d'entrée tandis que la CD ne réalise la déforestation que par une transformation symbolique indépendante de la méthode d'évaluation.

[6] La référence [11] n'est qu'une traduction non commentée.

Les limites actuelles de la composition descriptionnelle sont la nécessité de "visibilité" des constructeurs dans les règles sémantiques et l'impossibilité de traiter l'ordre supérieur (notion inhabituelle dans les domaines d'applications classiques des GAs). Cependant, la notion d'attribut hérité en GA permet de représenter des programmes qui nécessitent l'ordre supérieur dans le formalisme des *folds*. Les travaux sur l'AN, concernant initialement les *folds* du premier ordre, ont été étendus (et régulièrement améliorés) pour résoudre les problèmes posés par l'ordre supérieur (nécessaire aux *folds* pour atteindre un pouvoir d'expression suffisant). Il est intéressant de constater que la puissance d'expression induite par la notion d'attribut hérité permet de traiter la plupart des extensions nécessaires pour les *folds*. De plus, cette notion étant classique en GA, elle a été prise en compte initialement dans les nombreux travaux de recherche sur les GAs, et plus particulièrement concernant la CD.

Du point de vue pratique, des prototypes d'implantation de certaines de ces recherches ont été mis en place dans notre système de traitement de GA, FNC-2 [14] (la CD [22] et les GA Dynamiques [20]).

A la suite de cette présentation et cette première comparaison entre la CD et l'AN, purement intuitives et informelles, nous étudions actuellement comment formaliser plus précisément ces différentes notations (*folds* et GA) ainsi que ces méthodes de transformations (AN et CD), dans le but d'exhiber les possibilités de fertilisation croisée. Nous avons ainsi étendu la notion de généricité dirigée par la structure à la programmation fonctionnelle [7].

Bibliographie

[1] Alblas (Henk) et Melichar (Bořivoj) (édité par). – *Attribute Grammars, Applications and Systems.* – New York–Heidelberg–Berlin, Springer-Verlag, juin 1991, *Lect. Notes in Comp. Sci.*, volume 545. Prague.

[2] Boyland (John) et Graham (Susan L.). – Composing tree attributions. *In : 21st ACM Symp. on Principles of Programming Languages.* pp. 375–388. – Portland, Oregon, janvier 1994.

[3] Burstall (R. M.) et Darlington (John). – A transformation system for developing recursive programs. *Journal of the ACM*, vol. 24, n1, janvier 1977, pp. 44–67.

[4] 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).

[5] Duris (Etienne). – *Transformation de grammaires attribuées pour des mises à jour destructives.* – Rapport de DEA, Université d'Orléans, septembre 1994.

[6] Duris (Etienne), Parigot (Didier), Roussel (Gilles) et Jourdan (Martin). – *Attribute Grammars and Folds: Generic Control Operators.* – Rapport de recherche n 2957, INRIA, août 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2957.ps.gz`.

[7] Duris (Etienne), Parigot (Didier), Roussel (Gilles) et Jourdan (Martin). – Structure-directed genericity in functional programming and attribute grammars. – 1996. `ftp://ftp.inria.fr/INRIA/Projects/oscar/FNC-2/publications/Duris97gen.ps.gz`.

[8] Engelfriet (Joost). – Attribute grammars: Attribute evaluation methods. *In : Methods and Tools for Compiler Construction*, éd. par Lorho (Bernard), pp. 103–138. – New York, Cambridge University Press, 1984.

[9] Fegaras (Leonidas), Sheard (Tim) et Stemple (David). – Uniform traversal combinators: Definition, use and properties. *In : 11th International Conference on Automated Deduction (CADE-11).* pp. 148–162. – Saratoga Springs, New York, juin 1992.

[10] Fegaras (Leonidas), Sheard (Tim) et Zhou (Tong). – Improving programs which recurse over multiple inductive structures. *In : ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pp. 21–32. – Orlando, Florida, juin 1994.

[11] Fokkinga (M. M.), Jeuring (J.), Meertens (L.) et Meijer (E.). – A translation from attribute grammars to catamorphisms. *The Squiggolist*, vol. 2, n 1, 1991, pp. 20–26.

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

[13] 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.

[14] Jourdan (Martin), Parigot (Didier), Julié (Catherine), Durin (Olivier) et Le Bellec (Carole). – Design, implementation and evaluation of the FNC-2 attribute grammar system. *In : Conf. on Programming Languages Design and Implementation*, pp. 209–222. – White Plains, NY, juin 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[15] Knuth (Donald E.). – Semantics of context-free languages. *Mathematical Systems Theory*, vol. 2, n2, juin 1968, pp. 127–145. – Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[16] 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.

[17] 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.

[18] Paakki (Jukka). – Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, vol. 27, n2, juin 1995, pp. 196–255.

[19] Parigot (Didier) et Jourdan (Martin). – A bibliography on attribute grammars. – `http://www-rocq.inria.fr/oscar/FNC-2/AG.html` Updated regularly. Contains around 950 references to papers on Attribute Grammars. INRIA, France.

[20] 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.

[21] 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.

[22] Roussel (Gilles), Jourdan (Martin) et Parigot (Didier). – 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.

[23] 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.

[24] 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.

[25] van der Meulen (E. A.). – *Incremental Rewriting*. – Thèse de PhD, University of Amsterdam, 1994.

[26] Vogt (Harald H.), Swierstra (S. Doaitse) et Kuiper (Matthijs F.). – Higher order attribute grammars. *In : ACM SIGPLAN '89 Conf. on Progr. Lang. Design and Implementation*, pp. 131–145. – Portland, OR, juillet 1989. Published as *ACM SIGPLAN Notices*, 24(7).

[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.