Grammaires Attribuées et Folds : Opérateurs de Contrôle Génériques

Etienne Duris & Didier Parigot
& Gilles Roussel & Martin Jourdan

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, et en particulier l'élimination des structures intermédiaires "inutiles" apparaissant lors de la composition de fonctions. Une approche symbolique, la *déforestation*, a été proposée par Wadler [16] ; celle-ci était fondée sur les transformations de "pliage-dépliage" (ou "fold and unfold") introduites dans [2]. Depuis, différents formalismes ont été introduits pour étendre la puissance de ce type de transformations de programmes fonctionnels.

Nous nous sommes particulièrement intéressés à l'opérateur de contrôle générique *fold*, tel qu'il est présenté par Sheard et Fegaras [15] et qui est fondé sur un style de programmation *dirigé par la structure*. Cet opérateur 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 [9] et ont été largement étudiées depuis [1, 10, 12]. Une GA est une spécification déclarative qui décrit comment des attributs (des variables) sont calculés par des règles pour une grammaire particulière (i.e. programmation dirigée par la syntaxe). Les GAs ont été reconnues comme ayant ces 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 [7, 14]. Étant données deux GAs telles que le résultat de la première est l'argument de la seconde, la composition descriptionnelle permet de

Journées du GDR Programmation.
20, 21 et 22 novembre 1996.
Orléans.

produire une nouvelle GA qui a la même sémantique que la composition des GAs originales, mais qui ne construit plus la structure intermédiaire.

De plus, nous avons étendu la notion de GAs aux GAs Dynamiques [11, 13]. Notre vision de la grammaire sous-jacente à une GA est semblable à la grammaire qui décrit tous les arbres d'appels pour un programme fonctionnel donné, ou encore tous les arbres de preuve pour un programme logique donné : cette grammaire décrit précisément les différents flots de contrôle. Ceci permet d'utiliser les analyses statiques et les techniques d'implantation développées pour les GAs traditionnelles dans le contexte plus large des Schémas de Programmes Récursifs [3].

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. Cependant, nous ne traiterons ici que le cas des folds de premier ordre et des GAs purement synthétisées à un seul attribut. Une comparaison plus générale, étendue au second ordre et aux attributs hérités, a été faite dans [5].

La suite de cet article est divisée en deux sections. La première section présente 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* [15], ces derniers peuvent être considérés comme des évaluateurs particuliers pour les GAs purement synthétisées à un seul attribut. Dans la deuxième section, nous présentons 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 pratiquement équivalents (ils produisent les mêmes résultats).

2 Folds et Grammaires Attribuées

Les opérateurs de contrôle génériques *fold* sont définis sur des ensembles de *types mutuellement récursifs* [15]. Un tel type algébrique est défini par un ensemble de fonctions de construction (constructeurs) manipulant des variables de type. La Fig. 1 présente quelques exemples de définitions de types algébriques simples. Les GAs, qui sont traditionnellement spécifiées sur des grammaires non contextuelles (CFG), peuvent parfaitement 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. Par exemple, un constructeur de type algébrique $C_i(t_{i,1}, \ldots, t_{i,m_i})$ peut être considéré comme une 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 Fig. 1 sont présentées dans la Fig. 2.

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 (Fig. 3) et de concaténation de deux listes (Fig. 5) sont assez simples à exprimer [15]. Les deux lambda-expressions de la Fig. 3 sont appelées *fonctions accumulatives de résultat*

---

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

FIG. 1 – *Exemples de définitions de types algébriques*

Type list :	Type tree :	Type int :
Cons → α list	Node → tree tree	Succ → int
Nil →	Tip → α	Zero →

FIG. 2 – *Productions correspondant aux types de la Fig. 1*

```
length(x) = fold^list (   λ().Zero,
                          λ(a,r).Succ(r) ) x
```

FIG. 3 – *Définition de* length *avec un* fold

```
Nil ->
    f_{Nil,s↑Nil}      : s↑Nil = Zero            i.e. (λ().Zero)()
Cons ->a list
    f_{Cons,s↑Cons}    : s↑Cons = Succ(s↑list)   i.e. (λ(a,r).Succ(r))(a,s↑list)
```

FIG. 4 – *Définition de* length *en GA*

---

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 Déf. 2.1.

Nous considérons ici qu'une GA est la spécification, pour les productions (constructeurs) d'un type donné, d'un ensemble d'attributs, d'un ensemble de *règles sémantiques* permettant de calculer ces attributs pour chaque production et d'un *profil* (signature de la fonction représentée par la GA). Généralement, on distingue deux types d'attributs: les attributs synthétisés (qui permettent de remonter des calculs dans la structure de données) sont notés avec $\uparrow$ et les hérités (qui permettent de descendre des calculs dans la structure de données) avec $\downarrow$.

Sur les productions du type *list*, il est alors possible de décrire la fonction *length* avec une GA (Fig. 4). Les règles sémantiques $f_{Nil,s_{\uparrow Nil}}$ et $f_{Cons,s_{\uparrow Cons}}$ pour les productions (constructeurs) *Nil* et *Cons* dans la Fig. 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* (Fig. 3). Nous notons par exemple $f_{C,a_{\uparrow x}}$ la règle sémantique permettant de définir l'occurrence de l'attribut synthétisé $a_\uparrow$ 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 Fig. 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). Même si la comparaison [5] peut prendre en compte les attributs hérités (GAs générales) et les folds de second ordre (introduits dans [15]), nous nous contentons ici de comparer les GAs purement synthétisées (qui n'utilisent qu'un attribut synthétisé) avec les folds du premier ordre. Mais 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* (Fig. 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 (Fig. 5), peut également être spécifiée par une GA purement synthétisée (Fig. 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 par l'attribut $v_\uparrow$.

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

FIG. 5 – *Définition de* append(x,y) *avec un* fold

```
Nil ->
    f_{Nil,v↑Nil}      : v↑Nil = (λ().y)()
Cons ->a list
    f_{Cons,v↑Cons}    : v↑Cons = (λ(a,r).Cons(a,r))(a,v↑list)
```

FIG. 6 – *Définition de* append(x,y) *en GA*

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* [15]. Sans rentrer dans le détail du calcul de ces foncteurs, qui sont déterminés de façon statique à partir 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:*

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

Cette définition permet d'évaluer la fonction *length* de la Fig. 3, puisqu'elle permet d'identifier les paramètres de f_c (i.e. a et r). Le rôle du foncteur est donc 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*. La manière de calculer les fonctions accumulatives sur le type est définie par le foncteur. Ce dernier fournit 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*, défini une fois pour toute pour un type donné.

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 [6] 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 est indépendante de la méthode d'évaluation. Du point de vue des GAs, les foncteurs associés aux constructeurs d'un type peuvent être considérés comme une sorte d'évaluateur d'attributs.

Dans le cas particulier où une GA représente un programme exprimé en *fold*, les foncteurs spécifient un évaluateur d'attribut qui est correct pour la GA. De plus, la classe des GAs correspondant à des programmes exprimés en fold du premier ordre est bien connue. C'est en fait 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. Nous pouvons donc définir l'opérateur de contrôle générique $\text{fold}^{list}(f_n, f_c)$ avec la syntaxe des GAs sur les productions du type *list* (voir Fig. 7).

```
Nil ->
    f_{Nil,s↑_{Nil}}      : s↑_{Nil} = f_n
Cons ->a list
    f_{Cons,s↑_{Cons}}    : s↑_{Cons} = f_c(a,s↑_{list})
```

FIG. 7 – *GA générique pour le type* list

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))

FIG. 8 – Application à une Construction *sur* length

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 voulons maintenant comparer leurs facultés à éliminer des structures intermédiaires. Le but de cette partie est donc de comparer les méthodes de déforestation associées à chacun de ces formalismes : 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 sa 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 (Déf. 2.1) ; la Fig. 8 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 [15]. Ce théorème établit que la composition d'une fonction g avec une fonction exprimée en 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 :

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

Le Théorème de Promotion assure la validité du fold résultant pour des fonctions ϕ construites localement sur chaque constructeur, i.e. 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 [4] que l'algorithme de déforestation de Wadler [16] 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)* [15], nous allons dérouler chaque pas de l'AN. Tout d'abord, rappelons les définitions en fold de chacune de ces fonctions :

$$\begin{array}{l} \text{length(x)} = \text{fold}^{list}(\lambda().\text{Zero},\lambda(\text{a,r}).\text{Succ(r)})\ \text{x} \\ \text{append(x,y)} = \text{fold}^{list}(f_n, f_c)\ \text{x} \\ \quad \text{avec} \qquad\qquad f_n = \lambda().\text{y} \\ \qquad\qquad\qquad\ f_c = \lambda(\text{a,r}).\text{Cons(a,r)} \end{array}$$

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

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

L'*Application à une Construction* donne (d'après la Fig. 8) :

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

Et pour terminer, la *Généralisation* de $\text{length}(x_2)$ 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 :

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

dans lequel plus aucune structure intermédiaire n'est construite.

Composition Descriptionnelle

Considérons les types G_Ω, G_Δ et G_Θ. Pour deux GAs données, $\Omega(G_\Omega) \rightarrow G_\Delta$ et $\Delta(G_\Delta) \rightarrow G_\Theta$, le but de la CD est de construire une nouvelle GA $(\Delta \circ \Omega)(G_\Omega) \rightarrow G_\Theta$ 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 [5], et l'algorithme complet dans [7]). 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 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 Fig. 9.

Pour l'exemple de *length* $\circ$ *append*, considérons les définitions en GA de chacune des fonctions (Fig. 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 $f_{Cons,vs_{\uparrow_{Cons}}}$ (Fig. 9). 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

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↑Nil}      : vs↑Nil = (λ().length(y)) ()
Cons ->a list
    f_{Cons,vs↑Cons}    : vs↑Cons = (λ(a,r).Succ(r)) (a,vs↑list)
```

FIG. 9 – *La CD* (length ∘ append)(x,y) *de* (length(append(x,y)))

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 [2], i.e. le résultat de la CD de deux GAs non-circulaires est encore une GA non-circulaire [8].

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 [7] comme une sorte de Théorème de Promotion plus symbolique, au sens où la preuve de la correction de la CD est indépendante de la méthode d'évaluation des attributs [3] alors que le Théorème de Promotion est fortement lié à la définition des foncteurs. Nous affirmons donc que la CD est une 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.

4 Conclusion

Même si, à notre connaissance, cet article est la première tentative de comparaison entre les folds et les grammaires attribuées, ces formalismes révèlent déjà de nombreuses similitudes.

La première d'entre elles tient à la puissance d'expression des formes d'entrée, i.e. 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.

Dans cette brève présentation, nous n'avons pu présenter que le résultat le plus simple de cette comparaison. En effet, dans [5], nous avons également comparé le passage à l'ordre supérieur utilisé dans les folds avec l'utilisation naturelle des attributs hérité dans les GAs. La puissance d'expression obtenue est bien sûr plus importante, mais les méthodes d'élimination des structures intermédiaires, qui sont différentes, peuvent permettre une fertilisation croisée entre la programmation fonctionnelle et les grammaires attribuées.

2. Cette propriété de stabilité (clôture) des classes de GAs par la CD, principalement étudiée dans [8], n'est pas systématiquement réalisée pour toutes les classes.
3. En prenant garde cependant au problème de la clôture des classes de GAs par la CD.

Références

[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] 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.

[3] Courcelle (Bruno) et Franchi-Zannettacci (Paul). – Attribute grammars and recursive program schemes. *Theoretical Computer Science*, vol. 17, n2 and 3, 1982, pp. 163–191 and 235–257. – part I and II See also: rapport 8008, University de Bordeaux I (April 1980).

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

[5] 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`.

[6] 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.

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

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

[9] 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).

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

[11] Parigot (Didier), Duris (Étienne), Roussel (Gilles) et Jourdan (Martin). – *Attribute Grammars: a Declarative Functional Language.* – rapport de recherche n 2662, INRIA, octobre 1995. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2662.ps.gz`.

[12] Parigot (Didier) et Jourdan (Martin). – A bibliography on attribute grammars. – `http://www-rocq.inria.fr/oscar/FNC-2/AG.html` Updated regularly. Contains around 850 references to papers on Attribute Grammars. INRIA, France.

[13] Parigot (Didier), Roussel (Gilles), Jourdan (Martin) et Duris (Étienne). – Dynamic Attribute Grammars. *In : International Symposium on Programming Languages, Implementations, Logics and Programs.* – Aachen, septembre 1996.

[14] 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.

[15] Sheard (Tim) et Fegaras (Leonidas). – A fold for all seasons. *In : Conf. on Func. Prog. Languages and Computer Architecture.* pp. 233–242. – Copenhagen, Denmark, juin 1993.

[16] 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.