Programmation Polytypique avec les Grammaires Attribuées

Loïc Correnson

3 septembre 1997

Introduction

Ce travail s'inscrit dans le cadre d'un rapprochement entre la programmation fonctionnelle et les grammaires attribuées. Un des points de départ de cette démarche est la notion de déforestation :

Considérons l'expression $g(f(x))$. Pour en calculer la valeur, la fonction f construit une donnée intermédiaire qui est ensuite communiquée à g. La déforestation est l'opération qui consiste à fabriquer une fonction h équivalente à $g \circ f$ qui ne construit pas cette donnée intermédiaire, mais directement le résultat final.

Mais le point important est que cette déforestation est au cœur de la notion de grammaire attribuée générique. La généricité est un problème important pour la construction de grosses applications. Le but est de pouvoir réutiliser des fragments de programmes dans des situations variées.

L'an dernier, j'avais déjà étudié et approfondi cette notion de généricité dans les grammaires attribuées. Il était donc naturel de s'intéresser à une approche fonctionnelle, le polytypisme, une puissante extension du polymorphisme. Ma première approche a été d'exprimer le polytypisme en termes de grammaires attribuées. Ce travail a mis à jour la nécessité d'une forme d'évaluation partielle.

Or il se trouve que ce même outil est nécessaire pour traduire un programme fonctionnel en grammaire attribuée. Cette traduction est intéressante pour deux raisons. Premièrement, elle permet de réutiliser des programmes fonctionnels dans les grammaires attribuées. Deuxièmement elle nous permet de valider nos résultats sur la déforestation. En effet, par rapport aux approches de la programmation fonctionnelle, nous utilisons une méthode radicalement différente, appelée composition descriptionnelle [5, 6], qui est beaucoup plus performante. Pour ces raisons, mon travail s'est organisé en trois parties :

1. Définition de l'évaluation symbolique. C'est un mécanisme de projection qui permet, d'une part, de faire de l'évaluation partielle dans les grammaires attribuées, et d'autre part, de rendre explicite le flot des données d'un programme, ce qui est nécessaire à sa traduction en grammaire attribuée.

2. Reformulation de notre méthode de déforestation. Le but est de mettre en valeur ses relations avec l'évaluation symbolique, et de rendre cette transformation plus opérationnelle. Le résultat porte le nom de *composition symbolique*.

3. Comparaison des formes de généricité proposées par le polytypisme d'une

part et par les grammaires attribuées d'autre part. Le résultat de cette analyse est que le polytypisme est exprimable dans les grammaires attribuées, moyennant la création de nouveaux outils. Ainsi, sans changer notre notion fondamentale de généricité, nous en avons découvert un nouveau champ d'application.

Ce rapport est organisé de la manière suivante :

- Le chapitre 1 présente de manière succinte la notion de polytypisme et les grammaires attribuées. En tant que premier élément de comparaison, j'y décris aussi mon approche finale du polytypisme en grammaires attribuées.
- Le chapitre 2 présente la composition symbolique d'un point de vue assez technique. En effet, cette transformation est essentielle pour mettre en œuvre la généricité dans les grammaires attribuées.
- Dans le chapitre 3 est exposée une transcription primaire du polytypisme en termes de grammaires attribuées. Dans un deuxième temps, je simplifie le résultat obtenu afin d'aboutir à une forme finale plus intéressante du point de vue des grammaires attribuées.
- Enfin, le chapitre 4 présente notre forme de généricité et montre son aspect très général. En particulier, je montre comment on peut y intégrer le polymorphisme et le polytypisme.

Conventions

Quelques conventions de notations :

$\overline{u}$	: ensemble ou tuple composé de $u_1, \ldots, u_n$.
$\mathcal{E}[\ e\ ]$	: expression dans laquelle se trouve la sous-expression e.
$[x\ :=e]$	: substitution de la variable x par e.
$e : \tau$	: l'expression e est de type τ.
$\tau \rightarrow \tau'$	: type des fonctions de τ vers τ'.
$\tau \leftarrow \tau'$	: type des fonctions de τ' vers τ.
$a \stackrel{def}{=} e$	: définition locale dans un algorithme.
$C \vdash A \Rightarrow B$	: dans le contexte C, A est transformé en B.

Table des matières

1 Polytypisme **4**
1.1 `PolyP` 4
1.2 `AG` 7
1.3 Grammaires attribuées polytypiques 10

2 La composition symbolique **13**
2.1 Grammaires attribuées bien formées 13
2.2 Évaluation symbolique 14
2.2.1 La transformation 14
2.2.2 Evaluation partielle 16
2.2.3 Extraction de flot 16
2.3 Composition symbolique 17
2.3.1 Présentation intuitive 17
2.3.2 Algorithme 18
2.3.3 Application 20
2.3.4 Reformation d'appels 20
2.4 Élimination des règles de copie 22
2.5 Performance & comparaison 23

3 Généricité polytypique **25**
3.1 Copier `PolyP` dans `AG` 25
3.2 Vers une meilleure solution 26
3.3 Exemple 28

4 Généricité
dans les
grammaires attribuées **31**
4.1 Notion de couplage 31
4.2 Couplage par correspondance 32
4.3 Gène Optimal 33
4.4 Polytypisme et Couplage(s) 33
4.5 Polymorphisme et Couplage(s) 34

Chapitre 1

Polytypisme

1.1 PolyP

Cette section présente le plus succintement possible cette notion de généricité. Pour une présentation plus didactique, se référer à [8].

Intuitivement, une fonction polytypique est une fonction paramétrée par un type. On définit une fonction polytypique par induction sur la structure du type passé en paramètre, ou par composition avec d'autres fonctions polytypiques. Le langage `PolyP` est un sur-langage de Haskell qui permet d'écrire de telles fonctions.

On se place dans un système de types dans lequel les opérations suivantes sont définies :

- Somme : $\tau_1 + \tau_2$ représente la somme disjointe de τ_1 et τ_2.
- Produit : $\tau_1 * \tau_2$ représente le produit cartésien de τ_1 et τ_2.
- $\mathit{inl} : \tau_l \to \tau_l + \tau_r$: injection gauche.
- $\mathit{inr} : \tau_r \to \tau_l + \tau_r$: injection droite.
- $\mathit{fst} : \tau_1 * \tau_2 \to \tau_1$: première projection.
- $\mathit{snd} : \tau_1 * \tau_2 \to \tau_2$: deuxième projection.
- $\vartriangle : (\tau_1 \to \tau_1') \to (\tau_2 \to \tau_2') \to \tau_1 * \tau_2 \to \tau_1' * \tau_2'$: $f \vartriangle g$ applique f à la première composante d'une paire, et g à la deuxième.
- $\triangledown : (\tau_1 \to \tau) \to (\tau_2 \to \tau) \to (\tau_1 + \tau_2) \to \tau$: $f \triangledown g$ applique f sur une injection gauche, et g sur une injection droite.

Foncteurs et types réguliers

Un foncteur régulier est construit à partir de foncteurs de base, et des opérations de somme, produit et composition. Les foncteurs de base sont `Par` , pour le paramètre, `Rec` pour les composantes récursives, `Const` t pour un autre type t, `Empty` pour le type vide.

Définition 1.1.1 *Foncteur régulier :*

$$\mathcal{F}{::=}\texttt{Par} \mid \texttt{Rec} \mid \texttt{Const}\ t \mid \texttt{Empty} \mid \mathcal{F}_1 + \mathcal{F}_2 \mid \mathcal{F}_1 * \mathcal{F}_2 \mid \mathcal{F}_{comp}\ \mathcal{F}$$

Les actions d'un foncteur régulier sur les types et les morphismes de types sont très classiquement définies figure 1.1.

Définition 1.1.2 *Le type τ est régulier de foncteur $\mathcal{F}$ si, et seulement si :*

$$\tau \cong (\mathcal{F}\ \tau)$$

Existence du point fixe

Prenons un type abstrait τ paramétré par la variable de type a :

$$\tau\ a = \ldots \mid C_k\ \texttt{of}\ \tau_k^1 \ldots \tau_k^n \mid \ldots$$

Alors, on peut définir le foncteur $\mathcal{F}_\tau$ ainsi que les fonctions $\texttt{inn}_\tau$ et $\texttt{out}_\tau$ telles que :

$$\tau \overset{\texttt{out}_\tau}{\rightarrow} \mathcal{F}_\tau\ \tau \overset{\texttt{inn}_\tau}{\rightarrow} \tau$$

Voir figure 1.2 pour plus de détails. Ainsi, on montre que τ est régulier de foncteur $\mathcal{F}_\tau$.

Par exemple, pour le type :

$$\mathit{list}\ a = \mathit{cons}\ \texttt{of}\ a * \mathit{list}\ a | \mathit{nil}\ \texttt{of}\ ()$$

on a :

$$\begin{aligned}
&\mathcal{F}_{\mathit{list}} = \texttt{Par} * \texttt{Rec} + \texttt{Empty}\\
&\texttt{out}_{\mathcal{F}_{\mathit{list}}}\ (\mathit{cons}\ h\ t) = \mathit{inl}(h, t)\\
&\texttt{out}_{\mathcal{F}_{\mathit{list}}}\ \mathit{nil} = \mathit{inr}()\\
&\texttt{inn}_{\mathcal{F}_{\mathit{list}}} = \mathit{cons} \triangledown \mathit{nil}
\end{aligned}$$

Fonctions Polytypiques

Une fonction polytypique prend en paramètre le foncteur d'un type régulier. Elle peut être définie récursivement sur la structure de ce foncteur, comme pour la fonction $(\mathit{fmap}_{\mathcal{F}}\ f\ g)$ qui applique f sur les composantes correspondant à `Par` , et g sur celles correspondant à `Rec` :

Si f est un morphisme de type, et $\mathcal{F}$ un foncteur régulier, $(\mathcal{F}\ f)$ est le morphisme de type récursivement défini par :

$$\begin{array}{lcl}
\texttt{Rec}\ f & = & f \\
(\mathcal{F}_1 + \mathcal{F}_2)\ f & = & \lambda\theta.(\mathcal{F}_1\ f\ \theta)\ +\ (\mathcal{F}_2\ f\ \theta) \\
(\mathcal{F}_1 * \mathcal{F}_2)\ f & = & \lambda\theta.(\mathcal{F}_1\ f\ \theta)\ *\ (\mathcal{F}_2\ f\ \theta) \\
\mathcal{D}\,\mathcal{F}\ f & = & (\mathcal{D}\ (\lambda\theta.(\mathcal{F}\ f\ \theta))) \\
\mathcal{F}\ f & = & \lambda\theta.\theta \text{ dans les autres cas.}
\end{array}$$

Si τ est un type paramétré par a et $\mathcal{F}$ un foncteur régulier, $(\mathcal{F}\ \tau)$ est le type récursivement définit par :

$$\begin{array}{lcl}
\texttt{Par}\ \tau & = & a \\
\texttt{Rec}\ \tau & = & \tau \\
\texttt{Const}\ t\ \tau & = & t \\
\texttt{Empty}\ \tau & = & () \\
(\mathcal{F}_1 + \mathcal{F}_2)\ \tau & = & (\mathcal{F}_1\ \tau)\ +\ (\mathcal{F}_2\ \tau) \\
(\mathcal{F}_1 * \mathcal{F}_2)\ \tau & = & (\mathcal{F}_1\ \tau)\ *\ (\mathcal{F}_2\ \tau) \\
\mathcal{D}\,\mathcal{F}\ \tau & = & (\mathcal{D}\ \lambda\theta.(\mathcal{F}\ \theta))\ \tau
\end{array}$$

FIG. 1.1: *Définition d'un foncteur régulier*

Pour le type τ définit par :

$$\tau\ a = \ldots \mid C_k\ \texttt{of}\ \tau_k^1 \ldots \tau_k^n \mid \ldots$$

Définition de $\mathcal{F}_\tau$:

$$\begin{array}{lcl}
\mathcal{F}_\tau & = & \ldots + (\mathcal{F}_{\tau_k^1} * \ldots * \mathcal{F}_{\tau_k^n}) + \ldots \\
\mathcal{F}_{\tau_k^j} & = & \texttt{Par}\ \text{ si } \tau_k^j = a \\
 & = & \texttt{Rec}\ \text{ si } \tau_k^j = \tau\ a \\
 & = & \texttt{Const}\ t \text{ si } \tau_k^j = t \\
 & = & \texttt{Empty}\ \text{ si } \tau_k^j = () \\
 & = & \mathcal{F}_\theta\ \mathcal{F}_u \text{ si } \tau_k^j = \theta\ u
\end{array}$$

Définition de $\texttt{out}_\tau$:

$$\texttt{out}_\tau\ (C_k\ x^1 \ldots x^n) = \underbrace{(inl.inl\ldots)}_{k-1}.left(x^1, \ldots, x^n)$$

Définition de $\texttt{inn}_\tau$:

$$\texttt{inn}_\tau\ \underbrace{(inl.inl\ldots)}_{k-1}.left(y^1 \ldots y^n) = C_k\ y^1 \ldots y^n$$

FIG. 1.2: *Définition de $\mathcal{F}_\tau$, $\texttt{out}_\tau$ et $\texttt{inn}_\tau$*

$$
\begin{array}{l}
\texttt{polytypic}\ \mathit{fmap}_{\mathcal{F}} f\ g = \\
\quad \texttt{case}\ \mathcal{F}\ \texttt{of} \\
\qquad \begin{array}{lcl}
\texttt{Par} & \rightarrow & f \\
\texttt{Rec} & \rightarrow & g \\
\texttt{Const}\ t & \rightarrow & \texttt{id} \\
\texttt{Empty} & \rightarrow & \texttt{id} \\
(\mathcal{F}_1 + \mathcal{F}_2) & \rightarrow & \mathit{inl}.(\mathit{fmap}_{\mathcal{F}_1}\ f\ g) \triangledown \mathit{inr}.(\mathit{fmap}_{\mathcal{F}_2}\ f\ g) \\
(\mathcal{F}_1 * \mathcal{F}_2) & \rightarrow & (\mathit{fmap}_{\mathcal{F}_1}\ f\ g) \vartriangle (\mathit{fmap}_{\mathcal{F}_2}\ f\ g) \\
\mathcal{D}\,\mathcal{F} & \rightarrow & (\mathcal{D}\ (\mathit{fmap}_{\mathcal{F}}\ f\ g))
\end{array}
\end{array}
$$

On peut aussi définir une fonction polytypique par composition. Par exemple la fonction *pmap* qui applique une transformation sur tousles `Par` d'un arbre s'écrit :

$$\texttt{polytypic}\ \mathit{pmap}_{\mathcal{F}} = \mathtt{inn}_{\mathcal{F}}.(\mathit{fmap}_{\mathcal{F}}\ \varphi\ (\mathit{pmap}_{\mathcal{F}}\ \varphi)).\mathtt{out}_{\mathcal{F}}$$

Sur un type particulier, `PolyP` utilise un mécanisme d'évaluation partielle pour créer une instance de chaque fonction polytypique adaptée à ce type. Par exemple pour le type *list* on obtient :

$$
\begin{array}{l}
\mathit{fmap}_{\mathcal{F}_{list}}\ f\ g = \mathit{inl}.(f \vartriangle g) \triangledown \mathit{inr}.(\lambda x.x) \\
\mathit{pmap}_{list}\ \varphi = \mathtt{inn}_{list}.(\mathit{fmap}_{\mathcal{F}_{list}}\ \varphi\ (\mathit{pmap}_{list}\ \varphi)).\mathtt{out}_{list}
\end{array}
$$

Il faut alors recourir à des règles de simplification et utiliser les propriétés algébriques de $\vartriangle$ et $\triangledown$ pour en extraire la fonction suivante :

$$
\begin{array}{l}
\texttt{let}\ \mathit{pmap}_{list}\ \varphi\ l = \\
\quad \mathit{cons}\ h\ t \rightarrow (\mathit{cons}\ (\varphi\ h)\ (\mathit{pmap}_{list}\ t)) \\
\quad \mathit{nil} \rightarrow \mathit{nil}
\end{array}
$$

A ma connaissance, ce type de simplification n'est pas encore réalisé automatiquement dans `PolyP` .

1.2 AG

Définition

Avant toute chose, il faut noter que la définition des grammaires attribuées que nous utilisons ici n'est pas classique (voir aussi [16, 17]). Cependant, on retrouvera leurs caractéristiques essentielles, avec quelques libertés supplémentaires rendant ce paradigme de programmation plus opérationnel.

Les notions de base sont celles d'arbres, d'attributs et de décoration d'arbre.

Définition 1.2.1 *(arbre) Soit $\mathcal{C}$ un ensemble de constructeurs c d'arité $\sharp c$. On suppose qu'il existe au moins un constructeur d'arité nulle. L'ensemble des arbres sur $\mathcal{C}$ est défini par :*

$$t \in \mathcal{A} \Leftrightarrow \begin{cases} t \text{ est fini} \\ t = (c\ t_1 \dots t_{\sharp c}) \\ c \in \mathcal{C} \\ \forall k, t_k \in \mathcal{A} \end{cases}$$

En pratique, on utilise des constructeurs typés, et les arbres doivent être bien typés. On appèle aussi $\mathcal{C}$ une *grammaire.* Si $(c\ \bar{t})$ est un sous arbre de t, de type τ, on dit que c'est un nœud étiqueté par c, de type τ, dont les fils sont $t_1 \dots t_n$.

Définition 1.2.2 *(attribut, occurrences d'attributs) Pour tout type τ, on se donne un ensemble Att_τ d'attributs.*

Pour tout nœud x de type τ, et pour tout attribut a de Att_τ, on construit une nouvelle variable, notée $x.a$ et appelée occurrence de l'attribut a au nœud x.

Un arbre auquel on assigne une valeur à chaque occurrence d'attribut est un arbre décoré.

Maintenant, nous pouvons donner une définition d'une grammaire attribuée. Cette définition est une version très concise par rapport aux innombrables variantes que l'on peut rencontrer [15].

Définition 1.2.3 *(`AG`) Une grammaire attribuée $\mathcal{G}$ associe à tout constructeur c (de type $\tau_1 \dots \tau_n \to \tau$) un ensemble Σ_c d'équations entre les attributs de Att_τ, $Att_{\tau_1} \dots Att_{\tau_n}$.*

Soit t un arbre. On peut associer à tout nœud x de t étiqueté par c l'ensemble Σ_x des équations entre les occurrences d'attributs de x (et de ses fils) qui correspondent à Σ_c.

On construit ainsi Σ_t, le système d'équations obtenu en réunissant toutes les équations entre occurrences d'attributs.

La sémantique de $\mathcal{G}$ est l'application qui à t associe la solution de Σ_t si elle existe. Dans le cas contraire, cette application est indéfinie pour t.

Un exemple est présenté figure 1.3.

En pratique, on impose des restrictions sur les grammaires attribuées, afin de ne pas prendre en compte des systèmes d'équations trop généraux. Une description plus restrictive est donnée au début du chapitre 2.

De plus, dans la plupart des cas, on ne souhaite garder de la décoration d'un arbre qu'une seule occurrence d'attribut, en général située à la racine de l'arbre. Ainsi, une grammaire attribuée peut définir une fonction d'un arbre vers une de ses occurrences d'attributs.

Notations

Nous utilisons les notations de la figure 1.4 pour décrire une grammaire attribuée. Un *bloc* représente une grammaire attribuée unique. Les equations *eq* sont aussi appelées règles sémantiques.

Chaque déclaration de la forme $\{c\ \overline{x} \ \rightarrow\ eq*\}$ représente l'ensemble des équations associées à un nœud étiqueté par le constructeur c et ayant pour fils $x_1 \ldots x_n$. Les occurrences d'attribut du nœud $(c\ \overline{x})$ sont notées $this.a$, ou plus simplement a. Les occurrences d'attribut de ses fils sont notées $x_k.a$.

Nous avons signalé qu'une grammaire attribuée pouvait être vue comme une fonction. Dans cette perspective, il faut donc spécifier quel est l'occurrence d'attribut qui définit le résultat. De plus, dans la perspective d'un language de programmation réaliste, il faut pouvoir passer des paramètres lors d'un appel à une grammaire attribuée. C'est pourquoi la notion de profil est introduite :

La déclaration de la forme $\{f\ \overline{x} \rightarrow eq^*\}$ représente le profil de la fonction définie par la grammaire attribuée. Il est noté comme un cas particulier de constructeur, ce qui a deux intérêts. Premièrement, il définit quel est le résultat de cet appel. Par convention, on utilise l'attribut *result* qui n'a qu'une seule occurrence placée sous le profil. Deuxièmement, pour un arbre t, il complète le système d'équations Σ_t afin de tenir compte des paramètres d'appel.

Notons enfin que dans la pratique, on peut autoriser la déclaration de plusieurs profils pour la même grammaire attribuée.

Exemples

Voici par exemple comment on note la grammaire attribuée $\mathcal{L}$ de la figure 1.3 :

$\texttt{let}\ length =$	; nom de la grammaire attribuée
$\quad length\ x \rightarrow$	; profil de *length*
$\quad\quad result = x.length$	; *result* est le seul attribut du profil
	; la liste x a un attribut *length*
$\quad cons\ head\ tail \rightarrow$	; constructeur *cons* d'une liste
$\quad\quad length = (+\ 1\ tail.length)$	; definition de *length*
$\quad nil \rightarrow$	; constructeur *nil* d'une liste
$\quad\quad length = 0$	; definition de *length*

Un autre exemple un peu plus compliqué est *flatten*, qui construit à partir d'un arbre binaire, la liste de ses feuilles. On utilise pour celà un attribut $l^{flatten}$ comme accumulateur, et un attribut *flatten* pour calculer le résultat :

$$
\begin{array}{l}
\texttt{let}\ flatten = \\
\quad flatten\ t \rightarrow \\
\quad\quad result = t.flatten \\
\quad\quad t.l^{flatten} = nil \\
\quad node\ left\ right \rightarrow \\
\quad\quad flatten = right.flatten \\
\quad\quad right.l^{flatten} = left.flatten \\
\quad\quad left.l^{flatten} = l^{flatten} \\
\quad leaf\ n \rightarrow \\
\quad\quad flatten = cons\ n\ l^{flatten}
\end{array}
$$

Dans la suite, on utilise les abus de notations suivants :

– $(f\ \overline{t}).result$: cette expression correspond à appeler la grammaire attribuée f avec les paramètres $t_1 \dots t_n$. Notons que l'on peut aussi noter $(f\ \overline{t})$ cet appel. La distinction entre les deux notations intervient dans les problèmes d'évaluation partielle qui seront abordés au chapitre 2.

– $(c\ \overline{t}).a$: soit $\mathcal{G}$ la grammaire attribuée où est définit l'attribut a. Cette expression représente l'appel de $\mathcal{G}$ sur l'arbre $(c\ \overline{t})$. De la décoration ainsi calculée, on récupère alors la valeur de l'attribut a à la racine.

Notons que si on considère le profil f comme un constructeur particulier, le premier cas est un cas particulier du second.

1.3 Grammaires attribuées polytypiques

Le but de cette section est de montrer une grammaire attribuée qui permet d'exprimer la même notion de généricité que les fonctions polytypiques.

L'idée de base est de représenter un objet de n'importe quel type régulier τ par un arbre dont les constructeurs font référence au foncteur $\mathcal{F}_\tau$.

$$AG_{poly}\left\{\begin{array}{rl} sum : & \mathcal{P} \leftarrow \mathcal{C} * \mathcal{P} \\ prod : & \mathcal{P} \leftarrow \mathcal{P} * \mathcal{P} \\ empty : & \mathcal{P} \leftarrow () \\ rec : & \mathcal{P} \leftarrow \mathcal{P} \\ par : & \mathcal{P} \leftarrow A \\ const : & \mathcal{P} \leftarrow T \end{array}\right.$$

Par exemple, la liste d'un seul élement $(cons\ a\ nil)$ sera représentée par :

$$(rec\ (sum\ cons\ (prod\ (par\ a)\ (rec\ (sum\ nil\ empty)))))$$

Sur ce type, il est très facile d'écrire $pmap$:

$$\begin{array}{l} \texttt{let}\ pmap = \\ \quad pmap\ \varphi\ \mathcal{P} = \\ \quad\quad result = \mathcal{P}.pmap \\ \quad\quad \mathcal{P}.\varphi = \varphi \end{array}$$

$$\begin{array}{ll} sum\ c\ \mathcal{X} \rightarrow & \\ \quad pmap = (sum\ c\ \mathcal{X}.pmap) & rec\ \mathcal{P} \rightarrow \\ \quad \mathcal{X}.\varphi = \varphi & \quad pmap = \mathcal{P}.pmap \\ prod\ \mathcal{X}_1\ \mathcal{X}_2 \rightarrow & \quad \mathcal{P}.\varphi = \varphi \\ \quad pmap = (prod\ \mathcal{X}_1.pmap\ \mathcal{X}_2.pmap) & par\ A \rightarrow \\ \quad \mathcal{X}_1.\varphi = \varphi & \quad pmap = (par\ (\varphi\ A)) \\ \quad \mathcal{X}_2.\varphi = \varphi & const_t\ T \rightarrow \\ empty \rightarrow & \quad pmap = const_t\, T \\ \quad pmap = empty & \end{array}$$

Soit maintenant un type abstrait sur lequel on veut appliquer *pmap* :

$$\tau\ a = \ldots \mid C_k\ \texttt{of}\ \tau_k^1 \ldots \tau_k^n \mid \ldots$$

A partir de τ, on peut générer la grammaire attribuée $AG^{out}{}_\tau$ suivante :

$$
\begin{array}{l}
\texttt{let}\ AG^{out}{}_\tau = \\
\quad AG^{out}{}_\tau\ t \rightarrow \\
\qquad \mathit{result} = t.AG^{out}{}_\tau \\
\ldots \\
\quad C_k\ \bar{t} \rightarrow \\
\qquad AG^{out}{}_\tau = (\mathit{rec}\ (\mathit{sum}\ \ C_k\ \ (\mathit{prod}\ \ T_k^1\ \ (\mathit{prod}\ \ T_k^2\ \ \ldots T_k^n)))) \\
\ldots
\end{array}
$$

Avec, pour $T_k^1 \ldots T_k^n$ les définitions suivantes :

$$
\begin{array}{ll}
\text{Si } \tau_k^j = a & , T_k^j = \mathit{par}\ \ t_j \\
\text{Si } \tau_k^j = \tau & , T_k^j = t_j.AG^{out}{}_\tau \\
\text{Si } \tau_k^j = t & , T_k^j = \mathit{const}_t\ \ t_j
\end{array}
$$

Pour appliquer *pmap* sur τ il suffit alors de composer $AG^{in}{}_\tau$, *pmap* et $AG^{out}{}_\tau$. En utilisant la composition symbolique, on peut alors créer une grammaire attribuée $pmap_\tau$ qui ne construit pas de structure intermédiaire sur AG_{poly}.

Le chapitre 3 montre que l'on peut simuler plus fidèlement que AG_{poly} des fonctions polytypiques en utilisant deux grammaires. Mais on montre aussi que AG_{poly} est mieux adaptée aux fonctions génériques que l'on souhaite écrire avec des fonctions polytypiques.

On utilise la grammaire des listes :

$$list \begin{cases} cons: & list \leftarrow a * list \\ nil: & list \leftarrow () \end{cases}$$

On définit l'attribut suivant :

$$Att_{list} = \{l\}$$

La Grammaire attribuée $\mathcal{L}$, qui définit la longueur d'une liste, est définie comme suit :

$$\begin{array}{ll} \mathcal{L}(cons\ x\ y) & = \{l = +\ 1\ y.l\} \\ \mathcal{L}(nil) & = \{l = 0\} \end{array}$$

Prenons maintenant l'arbre t suivant (que l'on définit par ses nœuds) :

$$\begin{array}{ll} t = cons\ a\ x_1\ ; & x_1 = cons\ b\ x_2\ ; \\ x_2 = cons\ c\ x_3\ ; & x_3 = nil \end{array}$$

Les occurrences d'attributs présentes dans l'arbre sont donc :

$$Att(t) = \{t.l\ ;\ x_1.l\ ;\ x_2.l\ ;\ x_3.l\}$$

Le système Σ_t est le suivant :

$$\begin{array}{ll} t.l = +\ 1\ x_1.l\ ; & x_1.l = +\ 1\ x_2.l\ ; \\ x_2.l = +\ 1\ x_3.l\ ; & x_3.l = 0 \end{array}$$

La décoration de t par $\mathcal{L}$ est donc :

$$\mathcal{L}(t) = \{t.l = 3\ ;\ x_1.l = 2\ ;\ x_2.l = 1\ ;\ x_3.l = 0\}$$

FIG. 1.3: *Exemple de Grammaire Attribuée*

$$\begin{array}{lll} bloc & ::= & \texttt{let}\ f = \{f\ \overline{x} \rightarrow eq^*\}\{c\ \overline{x} \rightarrow eq^*\}^* \\ eq & ::= & occ = exp \\ occ & ::= & x.a \mid this.a \\ exp & ::= & Constant \\ & \mid & x \in Variables \\ & \mid & \lambda x.exp \\ & \mid & f\ \overline{exp} \\ & \mid & occ \end{array}$$

FIG. 1.4: *Notation des Grammaires Attribuées*

Chapitre 2

La composition symbolique

2.1 Grammaires attribuées bien formées

On a défini la sémantique d'une grammaire attribuée comme la résolution d'un système d'équations. Dans la pratique, pour calculer la décoration d'un arbre, on ne fait pas réellement cette résolution, ce qui prendrait trop de temps. On cherche plutôt à construire, indépendemment de tout arbre, un *évaluateur*, c'est à dire une stratégie de résolution qui permettra de calculer la décoration de tout arbre.

Ce type de résolution est très bien connu dans les grammaires attribuées, et correspond même à l'essentiel du travail de recherche mené jusqu'alors. Pour que la construction de l'évaluateur soit possible, on impose des restrictions sur la grammaire attribuée [1].

Classe des attributs

Si un nœud x est étiqueté par le constructeur c, on notera abusivement $Att(x)$ ou $Att(c)$ les occurrences d'attributs de x.

On partionne les attributs (Att) en deux sous-ensembles stricts : Att_S, l'ensemble des attributs synthétisés et Att_H, l'ensemble des attributs hérités. Intuitivement, les premiers correspondent à des valeurs qui remontent des feuilles d'un arbre vers sa racine, alors que les seconds redescendent dans l'arbre.

Plus précisement, on définit pour un constructeur c de fils $x_1 \ldots x_n$:

$$\begin{aligned} Att_{in}(c) &= \bigcup_i Att_S(x_i) \cup Att_H(c) \\ Att_{out}(c) &= \bigcup_i Att_H(x_i) \cup Att_S(c) \end{aligned}$$

Soit la grammaire attribuée $\mathcal{G}$:

$$\begin{array}{l} \texttt{let } \mathcal{G} = \\ \quad c\ \overline{x} \rightarrow \\ \quad\quad u = e \\ \quad \ldots \end{array}$$

1. Notons que la contrainte principale, non développée ici, est l'absence de circularité dans Σ_t.

$\mathcal{G}$ est bien formée si les conditions suivantes sont vérifiées :

C_1 : $u \in Att_{out}(c)$. On dit alors que l'occurrence u est définie.

C_2 : pour toute occurrence d'attribut v dans e, on a $v \in Att_{in}(c)$.

C_3 : toutes les occurrences d'attributs de $Att_{out}(c)$ sont définies une fois et une seule.

Inférences

Inférer la classe des attributs est très facile. Si la grammaire attribuée est bien formée, on peut faire la partition de Att en Att_S et Att_H en utlisant les points C_1 et C_2.

De plus, on peut s'autoriser quelques libertés concernant le point C_3. Grâce à la classe d'un attribut on peut en effet inférer un certain nombre d'équations. Par convention, si dans Att_{out}, une occurrence u de l'attribut a n'est pas définie, et si a n'a qu'une seule occurrence v dans Att_{in}, alors on infère l'équation $u = v$.

Par exemple, considérons la grammaire attribuée suivante :

$$\begin{array}{l}
\texttt{let}\ reverse = \\
\quad reverse\ x = \\
\quad\quad result = x.rev \\
\quad\quad x.l^{rev} = nil \\
\quad cons\ head\ tail \rightarrow \\
\quad\quad tail.l^{rev} = cons\ head\ l^{rev} \\
\quad nil \rightarrow \\
\quad\quad rev = l^{rev}
\end{array}$$

On en déduit facilement que $Att_S = \{s\}$ et $Att_H = \{h\}$. On a aussi :

$$\begin{array}{l}
Att_{in}(cons) = \{l^{rev},\ tail.rev\} \\
Att_{out}(cons) = \{rev,\ tail.l^{rev}\}
\end{array}$$

On peut donc inférer l'équation $s = tail.s$ qui manque pour que la grammaire attribuée soit bien formée. D'autres notions similaires permettent d'inférer encore plus de règles de copie.

En pratique, celà allège considérablement l'écriture d'une grammaire attribuée [9]. Les équations à écrire sont (presque) seulement celles où il y a une sémantique intéressante. De plus, le travail à fournir pour mettre à jour et maintenir de grosses grammaires attribuées en est considérablement réduit.

2.2 Évaluation symbolique

2.2.1 La transformation

Le but de cette transformation est d'éliminer les expressions de la forme $(c\ \overline{t}).a$. Soit donc un système Σ_0 où l'on trouve une telle expression.

$$\frac{\begin{pmatrix} c\ \overline{x} \rightarrow \\ \quad w = \varphi \\ \quad \Sigma_c \end{pmatrix} \in \mathcal{P} \qquad \begin{array}{l} \sigma \overset{def}{=} [x_i := t_i][this.h := e_h]_{\forall h} \\ \Sigma \overset{def}{=} \begin{cases} u = \mathcal{E}[\sigma(\varphi)] \\ \sigma(\Sigma_f) \\ \Sigma_{aux} \end{cases} \end{array} \qquad Check(c, d, \Sigma)}{\mathcal{P} \vdash \begin{pmatrix} d\ \overline{y} \rightarrow \\ \quad u = \mathcal{E}[(c\ \overline{t}).w] \\ \quad \{(c\ \overline{t}).h = e_h\}\forall h \\ \quad \Sigma_{aux} \end{pmatrix} \Rightarrow \begin{pmatrix} d\ \overline{y} \rightarrow \\ \Sigma \end{pmatrix}} \quad (SE)$$

$\mathcal{P} \vdash \Pi_1 \Rightarrow \Pi_2$ signifie : dans la grammaire attribuée $\mathcal{P}$, les équations Π_1 sont transformées en Π_2.

FIG. 2.1: *Évaluation Symbolique*

Si la grammaire attribuée est bien formée, alors les occurrences des attributs de $Att_{in}(c)$ sont définies. De plus, il n'y a pas de circularité dans les systèmes d'équations. Donc on est obligatoirement dans la situation suivante :

$$\Sigma_0 = \begin{cases} u = \mathcal{E}[\ (c\ \overline{t}).a\ ] \\ \{(c\ \overline{t}).h = e_h\} \quad \forall h \in Att_{in}(c) \\ \Sigma_{aux} \end{cases}$$

Soit maintenant Σ_c le système associé à c dans la grammaire attribuée, on peut remplacer $(c\ \overline{t}).a$ par l'expression qui définit a dans Σ_c. De plus, les occurrences des attributs h de $Att_{in}(c)$ peuvent être directement remplacées par les expressions e_h qui les définissent dans Σ_0.

Par exemple :

$$\begin{cases} u = (cons\ head\ tail).rev \\ (cons\ head\ tail).l^{rev} = \varphi \end{cases} \Rightarrow \begin{cases} u = tail.rev \\ tail.l^{rev} = (cons\ head\ \varphi) \end{cases}$$

Notons que les expressions de la forme $(f\ \overline{t}).result$ sont transformées de la même manière. D'où l'intérêt de noter les profils de fonctions comme des constructeurs particuliers.

Toutefois, cette projection n'est pas toujours possible. On tient notamment à obtenir une grammaire attribuée bien formée. On exprime cela par un prédicat *Check* qui dit si cette transformation est licite. Principalement, on vérifie que le système produit est bien formé (il doit respecter les conditions C_1, C_2 et C_3). On vérifie aussi que Σ_c n'est pas Σ_0 afin d'éviter une suite infinie de transformations. De plus, il convient d'avoir au préalable appliqué la transformation sur toutes les expressions $t_1 \ldots t_n$.

On obtient alors la règle de transformation définie figure 2.1.

2.2.2 Evaluation partielle

Un premier intérêt de l'évaluation symbolique est de faire de l'évaluation partielle. Prenons par exemple la grammaire attribuée qui calcule l'inversion d'une liste (voir section 2.1), et considérons le système :

$$\left\{ u = (reverse\ (cons\ a\ (cons\ b\ (cons\ c\ nil)))).result \right.$$

Une étape d'évaluation symbolique donne :

$$\left\{ \begin{array}{l} u = (cons\ a\ (cons\ b\ (cons\ c\ nil))).rev \\ (cons\ a\ (cons\ b\ (cons\ c\ nil))).l^{rev} = nil \end{array} \right.$$

Une autre étape donne :

$$\left\{ \begin{array}{l} u = (cons\ b\ (cons\ c\ nil)).rev \\ (cons\ b\ (cons\ c\ nil))).l^{rev} = (cons\ a\ nil) \end{array} \right.$$

Après deux autres étapes on obtiendra :

$$\left\{ \begin{array}{l} u = nil.rev \\ nil.l^{rev} = (cons\ c\ (cons\ b\ (cons\ a\ nil))) \end{array} \right.$$

Et donc finalement on obtiendra :

$$\left\{ u = (cons\ c\ (cons\ b\ (cons\ a\ nil))) \right.$$

On a donc fait de l'évaluation partielle.

2.2.3 Extraction de flot

Un deuxième intérêt de l'évaluation symbolique est de rendre explicite un flot de donnée implicite. Par exemple, imaginons que l'on écrive la grammaire attribuée *flatten* ainsi :

$$\begin{array}{l} \texttt{let}\ \mathit{flat} = \\ \quad \mathit{flat}\ t\ h \rightarrow \\ \quad\quad result = t.s \\ \quad\quad t.h = h \\ \quad node\ t_1\ t_2 \rightarrow \\ \quad\quad s = (\mathit{flat}\ t_2\ (\mathit{flat}\ t_1\ h).result).result \\ (\ldots) \end{array}$$

Le flot de données au niveau du constructeur *node* n'est pas apparent. En particulier, on ne voit pas comment les informations passent de t_1 à t_2. Une étape d'évaluation symbolique donne cependant :

$$\begin{array}{l} (\ldots) \\ \quad node\ t_1\ t_2 \rightarrow \\ \quad\quad s = (\mathit{flat}\ t_2\ t_1.s).result \\ \quad\quad t_1.h = h \\ (\ldots) \end{array}$$

Une deuxième étape donne :

$$\begin{array}{l} (\ldots) \\ \quad node\ t_1\ t_2 \rightarrow \\ \qquad s = t_2.s \\ \qquad t_2.h = t_1.s \\ \qquad t_1.h = h \\ (\ldots) \end{array}$$

Et l'on retrouve la définition désormais bien connue de *flatten*. Par rapport à la version initiale, la grammaire attribuée est bien formée et le flot des données est devenu explicite.

Notons au passage que l'équation :

$$s = (flat\ t_2\ (flat\ t_1\ h).result).result$$

initiale correspond à l'écriture de *flatten* dans un language fonctionnel comme Haskell. De fait, j'ai mis au point autour de l'évaluation symbolique une traduction d'un programme fonctionnel en une grammaire attribuée.

2.3 Composition symbolique

Ce qui suit est une présentation nouvelle de la déforestation dans les grammaires attribuée, aussi appelée *meta-composition*(article en soumisson [2]). Le premier apport est de rapprocher cette transformation d'une forme d'évaluation partielle, en utilisant l'évaluation symbolique pour éliminer les constructions intermédiaires. Par ailleurs, les points 2.3.3 et 2.3.4 donnent un aspect plus opérationnel à la transformation. Enfin dans cette approche, je me base sur des ensembles d'attributs pour savoir quoi et où projeter. Dans la *meta-composition* classique, on se basait sur la présence de constructeurs du type des données intermédiaires, ce qui posait des problèmes épineux.

2.3.1 Présentation intuitive

On veut composer deux grammaires attribuées. Pour le moment, peu importe pourquoi, ni comment cela à été déterminé (Cf. section 2.3.3).

Examinons quelle est la sémantique de $\mathcal{G} \circ \mathcal{F}$. Prenons un arbre t, et soit Σ_t le système d'équations que lui associe $\mathcal{F}$. Dans la solution de ce sytème se trouve un arbre t' sur lequel on doit appliquer $\mathcal{G}$.

Soit maintenant x' un nœud de t'. Ce nœud "provient" d'une équation associée à un nœud x de t. On sait donc, en examinant la grammaire attribuée $\mathcal{F}$, quels sont les endroits où la grammaire attribuée $\mathcal{G}$ devra être appliquée. Or $\mathcal{G}$ va décorer t'. Donc au nœud x' vont apparaître des occurrences attributs. Il est donc sémantiquement correct de projeter directement sur x ces occurrences. Toutefois, la grammaire attribuée ainsi obtenue peut ne pas être bien formée, auquel cas la composition échoue.

Exemple :

Prenons par exemple la composition de *reverse* et *flatten*. On veut projeter les occurrences des d'attributs *rev* et l^{rev} sur la liste créée par *flatten*. Nous nous intéressons dans un premier temps aux équations associées au constructeur *leaf* :

$$\begin{array}{l} leaf\ n \rightarrow \\ \quad this.flatten = cons\ n\ l^{flatten} \end{array}$$

Dans cette projection, on cherche bien sûr à obtenir une grammaire attribuée bien formée. Il faut donc tenir compte de la classe (hérité ou synthétisé) des attributs, et au besoin, ré-orienter les équations dans le bon sens. Ce qui donne :

$$\begin{array}{l} leaf\ n \rightarrow \\ \quad (this.flatten).rev = (cons\ n\ this.l^{flatten}).rev \\ \quad (cons\ n\ this.l^{flatten}).l^{rev} = (this.flatten).l^{rev} \end{array}$$

Maintenant, l'évaluation symbolique va éliminer la construction de la liste intermédiaire. En effet, rappelons que les équations associées au constructeur *cons* dans *reverse* sont :

$$\begin{array}{l} cons\ head\ tail \rightarrow \\ \quad this.rev = tail.rev \\ \quad tail.l^{rev} = cons\ head\ this.l^{rev} \end{array}$$

On obtiens donc :

$$\begin{array}{l} leaf\ n \rightarrow \\ \quad (this.flatten).rev = (this.l^{flatten}).rev \\ \quad (this.l^{flatten}).l^{rev} = (cons\ n\ (this.l^{flatten}).l^{rev}) \end{array}$$

On s'appercoit que l'élimination de la construction intermédiaire fait apparaître de nouvelles occurrences d'attributs, de la forme $(x.a).b$. L'étape finale consiste alors à les créer, on les note alors $x.a\_b$ pour $(x.a).b$.

2.3.2 Algorithme

La composition symbolique se décompose en trois étapes :

1. Projection des attributs,
2. Simplification des équations par évaluation symbolique,
3. Création des nouveaux attributs.

Avant de décrire ces étapes, une définition préliminaire est nécessaire.

Attributs Fonctionnels

Comme on l'a vu plus haut, il faut pouvoir isoler parmis les attributs qui participent aux calculs de $\mathcal{F}$, ceux qui construisent l'arbre destiné à $\mathcal{G}$. Pour celà

$$\frac{a \in Att_{\mathcal{F}} \quad \overline{s} = Att_S \cap Att_{\mathcal{G}} \quad \overline{h} = Att_H \cap Att_{\mathcal{G}}}{Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash x.a = \mathcal{E} \ \Rightarrow \left\{ \begin{array}{ll} (x.a).s_i = (\mathcal{E}).s_i & \forall i \\ (\mathcal{E}).h_j = (x.a).h_j & \forall j \end{array} \right.} \qquad (DC)$$

$Att_{\mathcal{G}}, Att_{\mathcal{F}} \vdash eq \ \Rightarrow \ \Sigma$ signifie que dans la composition $\mathcal{G} \circ \mathcal{F}$, l'équation eq (de $\mathcal{F}$) est transformée en l'ensemble d'équations Σ.

FIG. 2.2: *Etape de Projection*

il suffit de savoir quels sont, pour un attribut donné, les attributs nécessaires à son calcul[2].

Définition 2.3.1 *Pour tout attribut a, on appèle $Att_{fun}(a)$ le plus plus petit ensemble tel que :*

1. $a \in Att_{fun}(a)$
2. *Si une équation définit une occurrence d'attribut a_1 en fonction de a_2 alors :*
$$a_1 \in Att_{fun}(a) \quad \Rightarrow \quad a_2 \in Att_{fun}(a)$$

Si la grammaire attribuée est bien formée, alors on est sûr que les seuls attributs impliqués dans le calcul de a sont dans $Att_{fun}(a)$.

Projection

Quand on compose $\mathcal{F}$ et $\mathcal{G}$, on s'interesse en particulier à deux attributs $a_{\mathcal{F}}$ et $a_{\mathcal{G}}$ qui définissent respectivement le résultat de ces fonctions. On se place donc dans un contexte où l'on manipule deux ensembles d'attributs, définis comme suit :

$$Att_{\mathcal{F}} = Att_{fun}(a_{\mathcal{F}})\,; \quad Att_{\mathcal{G}} = Att_{fun}(a_{\mathcal{G}})$$

Avec ces notations, on définit la projection de $\mathcal{G}$ sur $\mathcal{F}$ comme une transformation des équations de $\mathcal{F}$, présentée figure 2.2.

Simplification

Une fois cette transformation appliquée, on doit appliquer l'évaluation symbolique autant que possible, afin d'éliminer toute construction intermédiaire. Cette transformation fait apparaître des termes de la forme $(\mathcal{E}.a).b$. On applique donc le "renommage" définit par la figure 2.3.

2. C'est une notion classique en grammaire attribuée, qui correspond aux calculs de dépendances servant à classifier les grammaires attribuées (NC, FNC...)

$$\frac{a \in att_{\mathcal{F}} \quad b \in Att_{\mathcal{G}}}{Att_{\mathcal{G}}, att_{\mathcal{F}} \vdash (\mathcal{E}.a).b \Rightarrow \mathcal{E}.a\_b}$$

$Att_{\mathcal{G}}, att_{\mathcal{F}} \vdash e_1 \Rightarrow e_2$ signifie qu'une expression e_1 de $\mathcal{G} \circ \mathcal{F}$ est transformée en l'expression e_2.

FIG. 2.3: *Etape finale de la Déforestation*

2.3.3 Application

Cette section a pour but de déterminer où doit être appliqué la composition symbolique. On peut identifier trois cas principaux. Chacun de ces cas est un motif particulier susceptible d'apparaître dans une expression :

$(g\ (f\ \overline{x}).result).result$:

On applique l'évaluation symbolique autant de fois que possible, ce qui élimine les $result$. Ce cas est donc eliminé. Toutefois, si l'évaluation symbolique n'est pas possible, on transforme cette expression dans la forme ci-dessous.

$(g\ (f\ \overline{x}))$:

Par convention, on ne fait aucune composition. Ceci permet de protéger des compositions que l'on ne souhaite pas faire, ou qui induisent des grammaires attribuées mal formées ou circulaires.

$(\mathcal{E}.a).b$:

On sait que l'on doit composer la grammaire attribuée qui calcule a avec celle qui calcule b. On applique alors l'algorithme de la figure 2.2 avec :

$$Att_{\mathcal{F}} = Att_{fun}(a) \quad ; \quad Att_{\mathcal{G}} = Att_{fun}(b)$$

En particulier, l'algorithme est appliqué à tous les systèmes où le motif $((\_.a).b)$ apparaît.

2.3.4 Reformation d'appels

Parfois, la grammaire attribuée peut rester malformée. Sitons deux cas qui peuvent se présenter et pour lesquels on a une solution :

- Il reste ou il apparaît des expressions de la forme $(\mathcal{E}.a).b$ qui ne sont pas éliminées. C'est une autre composition qui vient d'apparaître, et il suffit de refaire une composition symbolique.
- Il apparaît une expression $\mathcal{E}.a$ où $\mathcal{E}$ n'est pas un sous-arbre. Ce peut être le cas dans un profil de fonction si $\mathcal{E}$ est un paramètre de la fonction. Ce peut aussi être le cas si $\mathcal{E}$ est le résultat d'un appel de fonction dont on a pas la grammaire attribuée.

Le deuxième cas qui est plus compliqué à résoudre. Un exemple typique utilise la grammaire attribuée *append* qui concatène deux listes :

$$
\begin{array}{l}
\texttt{let}\ append = \\
\quad append\ a\ b \rightarrow \\
\qquad result = a.append \\
\qquad a.l^{append} = b \\
\quad cons\ head\ tail \rightarrow \\
\qquad append = cons\ head\ tail.append \\
\quad nil \rightarrow \\
\qquad append = l^{append}
\end{array}
$$

On cherche maintenant à faire l'inversion de la liste obtenue en concaténant a et b. On a donc :

$$
\begin{array}{l}
\varphi\ a\ b \rightarrow \\
\quad result = (reverse\ (append\ a\ b).result).result
\end{array}
$$

Une évaluation symbolique donne :

$$
\begin{array}{l}
\varphi\ a\ b \rightarrow \\
\quad result = (a.append).rev \\
\quad (a.l^{append}) = b \\
\quad (a.append).l^{rev} = nil
\end{array}
$$

On obtient après projection et simplification :

$$
\begin{array}{l}
\varphi\ a\ b \rightarrow \\
\quad result = a.append\_rev \\
\quad a.l^{append}\_rev = (b).rev \\
\quad (b).l^{rev} = a.l^{append}\_l^{rev} \\
\quad a.append\_l^{rev} = nil \\
cons\ head\ tail \rightarrow \\
(\ldots)
\end{array}
$$

Le problème se pose car b n'est pas un sous-arbre, φ etant un profil de fonction et non un constructeur d'arbre. Cependant, examinons le sous-système suivant :

$$
\left\{
\begin{array}{l}
a.l^{append}\_rev = (b).rev \\
(b).l^{rev} = a.l^{append}\_h
\end{array}
\right.
$$

L'expression $(b).rev$ siginifie qu'il faut appeler la grammaire attribuée qui décore b avec rev et l^{rev}, et retenir du résultat l'occurrence $b.rev$, tout en incorporant l'équation $b.l^{rev} = a.l^{append}\_l^{rev}$.

On sait que $Att_{fun}(rev) = \{rev\ ;\ h\}$. Donc on peut créer un nouveau profil pour la grammaire attribuée *reverse* :

$$
\begin{array}{l}
\varphi_{rev}\ L\ H \rightarrow \\
\quad result = L.rev \\
\quad L.l^{rev} = H
\end{array}
$$

Ainsi, le sous-système qui nous préoccupe est équivalent à :

$$a.l^{append}\_rev = (\varphi_{rev}\ \ b\ \ (a.l^{append}\_h))$$

Notons que l'appel à φ_{rev} n'est pas noté avec un $(\dots).result$ afin de protéger cet appel de toute future évaluation symbolique.

D'une manière générale, on sait à partir de Att_{fun} et du contexte dans lequel est appelée une grammaire attribuée, reconstituer un nouveau profil. Il permet d'appeler la grammaire attribuée sous forme d'un appel de fonction.

2.4 Élimination des règles de copie

Les grammaires attribuées générées par la composition symbolique peuvent contenir beaucoup de règles de copie inutiles. Celles-ci ne font que 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 [18].

C'est une étape très importante en pratique. Par exemple, aprè la composition de *reverse* avec *flatten*, seule la moitié des règles sont utiles. On passe ainsi de :

$$
\begin{array}{l}
\texttt{let}\ \mathit{flat} = \\
\quad \mathit{flat}\ t \rightarrow \\
\qquad \mathit{result} = t.\mathit{flat\_reverse} \\
\qquad t.\mathit{flat\_l}^{\mathit{reverse}} = \mathit{nil} \\
\qquad t.l^{\mathit{flat}}\_\mathit{reverse} = t.l^{\mathit{flat}}\_l^{\mathit{reverse}} \\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\qquad \mathit{flat\_reverse} = \mathit{right}.\mathit{flat\_reverse} \\
\qquad \mathit{right}.\mathit{flat\_l}^{\mathit{reverse}} = \mathit{flat\_l}^{\mathit{reverse}} \\
\qquad \mathit{right}.l^{\mathit{flat}}\_\mathit{reverse} = \mathit{left}.\mathit{flat\_reverse} \\
\qquad \mathit{left}.\mathit{flat\_l}^{\mathit{reverse}} = \mathit{right}.l^{\mathit{flat}}\_l^{\mathit{reverse}} \\
\qquad \mathit{left}.l^{\mathit{flat}}\_\mathit{reverse} = l^{\mathit{flat}}\_\mathit{reverse} \\
\qquad l^{\mathit{flat}}\_l^{\mathit{reverse}} = \mathit{left}.l^{\mathit{flat}}\_l^{\mathit{reverse}} \\
\quad \mathit{leaf}\ \ n \rightarrow \\
\qquad \mathit{flat\_reverse} = l^{\mathit{flat}}\_\mathit{reverse} \\
\qquad l^{\mathit{flat}}\_l^{\mathit{reverse}} = \mathit{cons}\ \ n\ \ \mathit{flat\_l}^{\mathit{reverse}}
\end{array}
$$

à la grammaire attribuée suivante :

$$
\begin{array}{l}
\texttt{let}\ \mathit{flat} = \\
\quad \mathit{flat}\ t \rightarrow \\
\qquad \mathit{result} = t.\mathit{flat\_rev} \\
\qquad t.\mathit{flat\_l}^{rev} = \mathit{nil} \\
\quad \mathit{node}\ \mathit{left}\ \mathit{right} \rightarrow \\
\qquad \mathit{flat\_rev} = \mathit{left}.l^{\mathit{flat}}\_l^{rev} \\
\qquad \mathit{right}.\mathit{flat\_l}^{rev} = \mathit{flat\_l}^{rev} \\
\qquad \mathit{left}.\mathit{flat\_l}^{rev} = \mathit{right}.l^{\mathit{flat}}\_l^{rev} \\
\quad \mathit{leaf}\ n \rightarrow \\
\qquad l^{\mathit{flat}}\_l^{rev} = \mathit{cons}\ n\ \mathit{flat\_l}^{rev}
\end{array}
$$

Cet exemple est très important d'un point de vue théorique. En effet il est le témoin d'une certaine classe de programmes, pour lesquels la composition symbolique s'applique là ou d'autres méthodes plus “fonctionnelles” échouent. Cet exemple montre aussi la capacité de la composition symbolique à optimiser le flot de données et à en adapter le contrôle. Ainsi, dans l'exemple ci-dessus, on a changé l'ordre dans lequel *flatten* traverse un arbre, pour construire la liste de ses feuilles en sens inverse.

2.5 Performance & comparaison

Déforestation classique

Dans la programmation fonctionnelle, la plupart des méthodes de déforestation dirigées par la structure sont basées sur des notions catégorielles comme les foncteurs, les catamorphismes et les hylomorphismes. Ces méthodes s'appuient sur des lois fondamentales telles que le *Théorème de Promotion* [13], et utilisent des opérateurs de contrôle génériques pour capturer simultanément le schéma de récursion des fonctions et celui des types réguliers. Tout d'abord, la *Short-cut Deforestation* [7] avec les règles d'élimination `foldr/buildr` a rendu ceci possible pour le type liste. Ensuite, l'*Algorithme de Normalisation* [19, 4, 11] a permis de généraliser la déforestation afin qu'elle fonctionne sur tous les types réguliers, grâce à la génération automatique de *foncteurs* à partir des définitions de types. Dernièrment les *hylomorphismes* [20] ont été introduits, mais la notion de foncteur est toujours utilisée pour représenter les appels récursifs de fonctions avec des foncteurs. Les systèmes tels que ADL et HYLO [14, 10] sont basés sur ce formalisme, et permettent de déforester certains programmes de manière automatisée.

Analyse

En dépit de ces raffinements successifs et de ces généralisations, une classe de programmes reste non déforestable par ces méthodes (e.g., $\mathit{reverse} \circ \mathit{reverse}$ et $\mathit{reverse} \circ \mathit{flatten}$ appartiennent à cette classe). De notre point de vue, les remarques informelles suivante permettent de caractériser en partie cette classe.

Les méthodes fonctionnelles utilisent toujours des foncteurs pour diriger les transformations et les calculs, ce qui ne s'avère pas toujours être le bon choix.

L'exemple de *reverse* est très représentatif :

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

Soit F_{rev} le foncteur qui dirige la récursion de cette fonction :

$$F_{rev} = \texttt{Rec}\ + (\texttt{Const}\ list)$$

Donc F_{rev} ne suit pas la construction de la liste produite par *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 $\underline{rev}$. Pour cette raison, la déforestation ne peut plus atteindre ce constructeur.

Dans les grammaires attribuées, la composition symbolique atteint chaque constructeur du résultat, directement à partir de la spécification. Cela ne nécessite aucune notion intermédiaire abstraite comme 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 symbolique est capable d'aboutir à plus de déforestation. Par exemple le $\underline{cons}$ ci-dessus est déforesté avec les grammaires attribuées.

Pour déforester de tels programmes, les approches fonctionnelles doivent utiliser un foncteur qui décrit complètement le schéma de construction du résultat, prenant en compte plus particulièrement les paramètres d'accumulations dans lesquels se font les constructions ou les transmissions des parties du résultat.

Chapitre 3

Généricité polytypique

3.1 Copier `PolyP` dans `AG`

Dans cette section, on cherche à montrer que les fonctions polytypiques peuvent être codées directement en grammaires attribuées habituelles. La première étape est de représenter les constructions élémentaires de somme et produit d'espace. On obtient ainsi la grammaire $\mathrm{AG}_{\mathcal{V}}$ d'un type universel :

$$\mathrm{AG}_{\mathcal{V}} \left\{ \begin{array}{l} \mathit{pair} \ : \mathcal{V} \leftarrow \mathcal{V} * \mathcal{V} \\ \mathit{left} \ : \mathcal{V} \leftarrow \mathcal{V} \\ \mathit{right} \ : \mathcal{V} \leftarrow \mathcal{V} \\ \mathit{object} \ : \mathcal{V} \leftarrow T \end{array} \right.$$

Dans cette définition, T représente n'importe quel type, et le constructeur *object* sera souvent omis.

Maintenant, nous pouvons donner la définition en grammaire attribuée de l'opérateur $\triangledown$, que nous notons ici comme une fonction :

$$\begin{array}{l} \triangledown \ f \ g \ \mathcal{V} \rightarrow \\ \quad \mathit{result} = \mathcal{V}.\triangledown \\ \quad \mathcal{V}.\triangledown_f = f \\ \quad \mathcal{V}.\triangledown_g = g \\ \mathit{left} \ \mathcal{V} \rightarrow \\ \quad \triangledown = (\triangledown_f \ \mathcal{V}) \\ \mathit{right} \ \mathcal{V} \rightarrow \\ \quad \triangledown = (\triangledown_g \ \mathcal{V}) \end{array}$$

Pour $\vartriangle$, on trouve une définition similaire. La deuxième étape est de représenter les foncteurs de type. On utilise pour cela la grammaire suivante :

$$\mathrm{AG}_{\mathcal{F}} \begin{cases} sum_{\mathcal{F}} : & \mathcal{F} \leftarrow \mathcal{F} * \mathcal{F} \\ prod_{\mathcal{F}} : & \mathcal{F} \leftarrow \mathcal{F} * \mathcal{F} \\ empty_{\mathcal{F}} : & \mathcal{F} \leftarrow () \\ rec_{\mathcal{F}} : & \mathcal{F} \leftarrow () \\ par_{\mathcal{F}} : & \mathcal{F} \leftarrow () \\ const_{\mathcal{F}} : & \mathcal{F} \leftarrow () \\ comp_{\mathcal{F}} : & \mathcal{F} \leftarrow \mathcal{F} * \mathcal{F} \end{cases}$$

On peut générer les grammaires attribuées `out` et `inn` de la même façon que pour les fonctions polytypiques. On obtiens par exemple, pour le type *list* :

$$\begin{array}{l} \mathcal{F}_{list} = (sum_{\mathcal{F}}\ (prod_{\mathcal{F}}\ par_{\mathcal{F}}\ rec_{\mathcal{F}})\ empty_{\mathcal{F}}) \\ \texttt{let out} = \\ \quad \texttt{out}\ l \rightarrow \\ \quad\quad result = l.\texttt{out} \\ \quad cons\ h\ t \rightarrow \\ \quad\quad \texttt{out} = left\ (pair\ (object\ h)\ (object\ t)) \\ \quad nil \rightarrow \\ \quad\quad \texttt{out} = right\ (object()) \end{array}$$

3.2 Vers une meilleure solution

A propos de `out`

Une première remarque s'impose : `out` ne transforme que le constructeur de tête d'un arbre. Cependant, la plupart des algorithmes qui sont codés par des fonctions polytypiques vont inspecter récursivement tout ou partie d'un arbre. Ainsi *pmap* appelle récursivement `out` sur tous les nœuds d'un arbre.

Dans cette optique, on est obligé d'écrire *pmap* en sachant que la fonction va travailler sur un type qui n'est pas le type universel, et qu'il sera nécessaire de faire des appels à `out` pour se placer dans les bonnes conditions de travail.

Les grammaires attribuées offrent la possibilité d'une approche un peu différente, où l'on cherche à écrire un algorithme générique sur le type le plus approprié, sans se soucier du type concret qui sera effectivement utilisé.

Dans cette optique, `out` doit être remplacée par une version entièrement récursive :

$$\texttt{out}_{ag}\ \mathcal{F} = (fmap\ \mathcal{F}\ \texttt{id}\ \ \texttt{out}_{ag}) \circ \texttt{out}$$

Et dans ce cadre, on a :

$$pmap\ \mathcal{F}\ \varphi = (fmap\ \mathcal{F}\ \varphi\ (pmap\ \mathcal{F}\ \varphi))$$

avec aucune référence à `out`. Par contre, *pmap* ne s'applique que sur des arbres de $\mathrm{AG}_{\mathcal{V}}$.

Vers la définition de AG_{poly}

On a utilisé précédemment deux grammaires, $\mathrm{AG}_{\mathcal{F}}$ pour décrire les foncteurs des types abstraits, et $\mathrm{AG}_{\mathcal{V}}$ pour décrire des valeurs dans un type universel. Mais en fait, une notion de typage relie fortement ces deux grammaires. Si x est de type τ de foncteur $\mathcal{F}_\tau$, alors le constructeur de tête de x correspond à celui de $\mathcal{F}_\tau$. Ainsi, à une somme correspond une injection, à un produit correspond une paire, etc.

Finalement, on peut tout à fait coder dans le type universel l'information qui se trouve dans le foncteur. On peut donc utiliser la grammaire suivante, qui ne sera pas notre version définitive, mais seulement une première simplification de $\mathrm{AG}_{\mathcal{F}}$ et $\mathrm{AG}_{\mathcal{V}}$:

$$AG_{poly}\left\{\begin{array}{rl} sum_l : & \mathcal{P} \leftarrow \mathcal{P} \\ sum_r : & \mathcal{P} \leftarrow \mathcal{P} \\ prod : & \mathcal{P} \leftarrow \mathcal{P} * \mathcal{P} \\ empty : & \mathcal{P} \leftarrow \\ rec : & \mathcal{P} \leftarrow \mathcal{P} \\ par : & \mathcal{P} \leftarrow A \\ const : & \mathcal{P} \leftarrow T \end{array}\right.$$

Le cas de la composition de foncteurs est fortement lié à la notion de polymorphisme, et il est le traîté à la section 4.5. En fait l'utilisation des grammaires attribuées permet de se passer de cette composition.

Sommes

Cherchons maintenant quel type d'opérations sémantiques on peut spécifier pour les constructeurs sum_l et sum_r. Considérons la fonction polytypique suivante, et son homologue en grammaire attribuée sur AG_{poly} :

$\texttt{polytypic}\ f\ \mathcal{F} \ldots$ $\quad\texttt{case}\ \mathcal{F}\ \texttt{of}$ $\qquad \mathcal{F}_1 + \mathcal{F}_2 \rightarrow \varphi \triangledown \psi$ $\qquad \ldots$	$\texttt{let}\ f\ \mathcal{F} \ldots$ $\quad \ldots$ $\quad sum_l\ \mathcal{F}_l \rightarrow$ $\qquad f = \varphi\ \mathcal{F}_l$ $\quad sum_r\ \mathcal{F}_r \rightarrow$ $\qquad f = \psi\ \mathcal{F}_l$ $\quad \ldots$

Maintenant, instancions f :

- sur le type $\tau = C_1|C_2$: f va appliquer φ sur C_1 et ψ sur C_2.
- sur le type $\tau' = C_2|C_1$: f va appliquer φ sur C_2 et ψ sur C_1.

A l'évidence, $\tau = \tau'$. Donc, pour que l'instanciation ait une valeur sémantique interessante, il faut que $\psi = \varphi$, où bien il faut que l'opération voulue dépende explicitement du constructeur utilisé.

En fait, les constructeurs sum_l et sum_r forment un codage des constructeurs du type τ. Il est donc plus simple de ne garder qu'un seul constructeur sum, avec comme paramètre supplémentaire le constructeur auquel il est associé.

On arrive donc à la solution finale :

$$AG_{poly} \left\{ \begin{array}{rl} sum : & \mathcal{P} \leftarrow \mathcal{C} * \mathcal{P} \\ prod : & \mathcal{P} \leftarrow \mathcal{P} * \mathcal{P} \\ empty : & \mathcal{P} \leftarrow () \\ rec : & \mathcal{P} \leftarrow \mathcal{P} \\ par : & \mathcal{P} \leftarrow A \\ const : & \mathcal{P} \leftarrow T \end{array} \right.$$

qui est celle présentée dans le premier chapitre.

3.3 Exemple

Nous montrons par exemple l'instanciation d'une fonction polytypique *pflat* qui produit la liste des objets de type a dans un arbre de type $\tau\ a$. Cette liste est construite à l'aide d'un accumulateur véhiculé par les attributs s et h :

$$\begin{array}{l} \texttt{let}\ \mathit{pflat} = \\ \quad \mathit{pflat}\ \mathcal{P} \rightarrow \\ \qquad \mathit{result} = \mathcal{P}.s \\ \qquad \mathcal{P}.h = \mathit{nil} \\ \quad \mathit{prod}\ \mathcal{P}_1\ \mathcal{P}_2 \rightarrow \\ \qquad \mathcal{P}_1.h = h \\ \qquad \mathcal{P}_2.h = \mathcal{P}_1.s \\ \qquad s = \mathcal{P}_2.s \\ \quad \mathit{par}\ \ a \rightarrow \\ \qquad s = \mathit{cons}\ \ a\ \ h \end{array}$$

Les autres équations étant automatiquement inférées. On veut maintenant instancier cette fonction sur le type des arbres binaires, décris par la grammaire suivante :

$$\left\{ \begin{array}{ll} node : & Tree \leftarrow Tree * Tree \\ leaf : & Tree \leftarrow A \end{array} \right.$$

Composition

On veut donc composer *pflat* avec *out*, ce qui donne :

$$\begin{array}{l} \texttt{let}\ \mathit{pflat}_{Tree} = \\ \quad \mathit{pflat}_{Tree}\ \ t \rightarrow \\ \qquad \mathit{result} = (\mathit{pflat}\ \ t.\mathit{out}).\mathit{result} \\ \texttt{let}\ \mathit{out} = \\ \quad \mathit{node}\ \ t_1\ \ t_2 \rightarrow \\ \qquad \mathit{this}.\mathit{out} = \mathit{rec}\ (\mathit{sum}\ \ \mathit{node}\ \ (\mathit{prod}\ \ t_1.\mathit{out}\ \ t_2.\mathit{out})) \\ \quad \mathit{leaf}\ \ a \rightarrow \\ \qquad \mathit{this}.\mathit{out} = \mathit{rec}\ (\mathit{sum}\ \ \mathit{leaf}\ \ (\mathit{par}\ \ a)) \end{array}$$

Une étape d'évaluation symbolique donne :

$$
\begin{array}{l}
\texttt{let } \mathit{pflat}_{\mathit{Tree}} = \\
\quad \mathit{pflat}_{\mathit{Tree}}\ t \rightarrow \\
\quad\quad \mathit{result} = (t.\mathit{out}).\mathit{pflat} \\
\quad\quad (t.\mathit{out}).h = \mathit{nil}
\end{array}
$$

Projection

En utilisant les notations du chapitre 2, on a :

$$\mathit{Att}_{\mathit{fun}}(\mathit{out}) = \{\mathit{out}\} \quad \mathit{Att}_{\mathit{fun}}(\mathit{pflat}) = \{\mathit{pflat}\,;h\}$$

La projection donne :

$$
\begin{array}{l}
\texttt{let } \mathit{pflat}_{\mathit{Tree}} = \\
\quad \mathit{pflat}_{\mathit{Tree}}\ t \rightarrow \\
\quad\quad \mathit{result} = (t.\mathit{out}).\mathit{pflat} \\
\quad\quad (t.\mathit{out}).h = \mathit{nil} \\
\quad \mathit{node}\ t_1\ t_2 \rightarrow \\
\quad\quad (\mathit{this}.\mathit{out}).\mathit{pflat} = (\mathit{rec}\ (\mathit{sum}\ \mathit{node}\ (\mathit{prod}\ t_1.\mathit{out}\ t_2.\mathit{out}))).\mathit{pflat} \\
\quad\quad (\mathit{rec}\ (\mathit{sum}\ \mathit{node}\ (\mathit{prod}\ t_1.\mathit{out}\ t_2.\mathit{out}))).h = (\mathit{this}.\mathit{out}).h \\
\quad \mathit{leaf}\ a \rightarrow \\
\quad\quad (\mathit{this}.\mathit{out}).\mathit{pflat} = (\mathit{rec}\ (\mathit{sum}\ \mathit{leaf}\ (\mathit{par}\ a))).\mathit{pflat} \\
\quad\quad (\mathit{rec}\ (\mathit{sum}\ \mathit{leaf}\ (\mathit{par}\ a))).h = (\mathit{this}.\mathit{out}).h
\end{array}
$$

Evaluation symbolique

Après de multiples étapes d'évaluation symbolique on obtient :

$$
\begin{array}{l}
\texttt{let } \mathit{pflat}_{\mathit{Tree}} = \\
\quad \mathit{pflat}_{\mathit{Tree}}\ t \rightarrow \\
\quad\quad \mathit{result} = (t.\mathit{out}).\mathit{pflat} \\
\quad\quad (t.\mathit{out}).h = \mathit{nil} \\
\quad \mathit{node}\ t_1\ t_2 \rightarrow \\
\quad\quad (\mathit{this}.\mathit{out}).\mathit{pflat} = (t_2.\mathit{out}).\mathit{pflat} \\
\quad\quad (t_2.\mathit{out}).h = (t_1.\mathit{out}).\mathit{pflat} \\
\quad\quad (t_1.\mathit{out}).h = (\mathit{this}.\mathit{out}).h \\
\quad \mathit{leaf}\ a \rightarrow \\
\quad\quad (\mathit{this}.\mathit{out}).\mathit{pflat} = \mathit{cons}\ a\ (\mathit{this}.\mathit{out}).h
\end{array}
$$

Simplification

Conformément à $\mathit{Att}_{\mathit{fun}}(\mathit{out})$ et $\mathit{Att}_{\mathit{fun}}(\mathit{pflat})$, on obtient pour finir :

$$
\begin{array}{l}
\texttt{let}\ \mathit{pflat}_{\mathit{Tree}} = \\
\quad \mathit{pflat}_{\mathit{Tree}}\ t \rightarrow \\
\quad\quad \mathit{result} = t.\mathit{out\_pflat} \\
\quad\quad t.\mathit{out\_h} = \mathit{nil} \\
\quad \mathit{node}\ t_1\ t_2 \rightarrow \\
\quad\quad \mathit{this}.\mathit{out\_pflat} = t_2.\mathit{out\_pflat} \\
\quad\quad t_2.\mathit{out\_h} = t_1.\mathit{out\_pflat} \\
\quad\quad t_1.\mathit{out\_h} = \mathit{this}.\mathit{out\_h} \\
\quad \mathit{leaf}\ a \rightarrow \\
\quad\quad \mathit{this}.\mathit{out\_pflat} = \mathit{cons}\ a\ \mathit{this}.\mathit{out\_h}
\end{array}
$$

Remarquons que l'on obtient ainsi automatiquement la grammaire attribuée *flatten* définie à la section 1.2.

Chapitre 4

Généricité dans les grammaires attribuées

4.1 Notion de couplage

Dans les grammaires attribuées, il existe une notion de généricité très simple, dont le coeur est l'utilisation massive de la composition symbolique [3]. Dans ce cadre, un algorithme générique $\mathcal{A}_g$ est une grammaire attribuée tout à fait classique, définie sur une grammaire $\mathcal{C}_g$.

Soit maintenant une grammaire $\mathcal{C}_p$ particulière sur laquelle on veut instancier l'algorithme $\mathcal{A}_g$. On suppose que l'on est capable de définir une grammaire attribuée $\mathcal{A}_{p\to g}$, dite "de couplage", qui extrait d'un arbre de $\mathcal{C}_p$, l'arbre de $\mathcal{C}_g$ qui lui correspond. Notons que cette correspondance existe implicitement à partir du moment où l'on souhaite instancier $\mathcal{A}_g$ sur $\mathcal{C}_p$. L'algorithme instancié est alors tout simplement :

$$\mathcal{A}_p \overset{def}{=} \mathcal{A}_g \circ \mathcal{A}_{p\to g}$$

La composition symbolique se charge alors d'effectuer l'instanciation, et d'en founir une version deforestée par soucis d'efficacité.

Par ailleurs, cette approche permet la réutilisation de code de manière très efficace. En effet, comme un algorithme générique est une grammaire attribuée normale, on peut réutiliser un algorithme déjà existant.

A partir de cette notion de généricité très générale, deux questions restent ouvertes :

- Dans quelle mesure peut-on générer automatiquement $\mathcal{A}_{p\to g}$? Bien entendu, on peut écrire à la main ces couplages, mais dans la pratique, on souhaite un moyen concis de déclarer de telles grammaires.

- Comment choisir $\mathcal{C}_g$? Cette question est bien entendu en étroite relation avec la première. Le but étant de permettre une instanciation automatique dans le plus grand nombre de cas possible.

4.2 Couplage par correspondance

Dans [12], nous proposons une réponse possible à la première question. On introduit la notion de correspondance entre deux grammaires :

Définition 4.2.1 *Cor est une relation de correspondance de la grammaire* $\mathcal{C}_p$ *vers la grammaire* $\mathcal{C}_g$*, si et seulement si :*

- *Cor associe à un type* τ *de* $\mathcal{C}_g$ *un ensemble de types* $Cor(\tau)$ *de* $\mathcal{C}_p$*.*
- *si* $\tau \neq \tau'$*, alors* $Cor(\tau) \cap Cor(\tau') = \emptyset$*.*

Avec cette définition, on peut inférer dans de nombreux cas des grammaires attribuées de couplage. Pour cela on cherche à associer un morceau d'arbre de $\mathcal{C}_p$ avec un constructeur de $\mathcal{C}_g$. Prenons par exemple la grammaire $\mathcal{C}_p$ suivante :

$$\begin{array}{ll}
prog_dec & : prog \leftarrow decl \ * \ prog \\
prog_end & : prog \leftarrow \\
decl_var & : decl \leftarrow var \ * \ def \\
def_option & : def \leftarrow option \ * \ def \\
def_exp & : def \leftarrow exp
\end{array}$$

On souhaite faire un couplage vers la grammaire des listes, pour récupérer la liste des *exp* d'un arbre de type *prog*. On donne donc la correspondance :

$$\begin{array}{l}
Cor(List) = \{prog\} \\
Cor(A) = \{exp\}
\end{array}$$

On peut alors inférer automatiquement la grammaire attribuée $\mathcal{A}_{p\rightarrow List}$ suivante :

$$\begin{array}{l}
\mathcal{A}_{p\rightarrow List}\ p \rightarrow \\
\quad result = t.list \\
prog_dec\ \ d\ \ p \rightarrow \\
\quad list = (cons\ \ d.a\ \ p.list) \\
prog_end \rightarrow \\
\quad list = nil \\
decl_var\ \ v\ \ d \rightarrow \\
\quad a = d.a \\
def_option\ \ o\ \ d \rightarrow \\
\quad a = d.a \\
def_expe \rightarrow \\
\quad a = e
\end{array}$$

Dans [1] je présente les algorithmes complets et la réalisation d'un prototype de cette transformation. On remarquera que les objets de type *var* et *option* qui précèdent une expression sont automatiquement laissés de côté.

4.3 Gène Optimal

Dans [1], j'ai exploré la réponse à la deuxième question dans ce cadre particulier des couplages par correspondance. Il y a souvent plusieurs grammaires possibles pour décrire un algorithme générique. Parmi celles-ci, on en distingue une, dite "en forme standard", qui permet la génération automatique d'un couplage par correspondance dans un nombre de cas maximum. De plus, il existe un algorithme qui calcule, à partir d'une grammaire $\mathcal{C}_0$ les deux objets suivants :

- Une grammaire $\mathcal{C}_g$ en forme standard qui est isomorphe à $\mathcal{C}_0$.
- Une grammaire attribuée $\mathcal{A}_{0\to g}$ de couplage de $\mathcal{C}_0$ vers $\mathcal{C}_g$.

Ceci permet de réutiliser efficacement du code. En effet, prenons une grammaire attribuée $\mathcal{A}_0$, dont on souhaite réutiliser le code. On définit son algorithme générique $\mathcal{A}_g$ ainsi :

$$\mathcal{A}_g = \mathcal{A}_{0\to g} \circ \mathcal{A}_g$$

On obtient ainsi un algorithme générique que l'on peut instancier par relation de correspondance sur un plus grand nombre de cas que pour $\mathcal{A}_0$.

4.4 Polytypisme et Couplage(s)

Notre point de vue sur la notion de polytypisme s'exprime comme une forme particulière de couplage. En effet, on peut construire AG^{out} (et AG^{in}) à partir d'une *nouvelle* forme de correspondance, par exemple :

$$\begin{array}{l} Cor_{\texttt{PolyP}}\ (Par) = \{A\} \\ Cor_{\texttt{PolyP}}\ (Rec) = \{List\} \end{array}$$

Remarquons que AG_{poly} permet une instanciation automatique sur toute les grammaires. Cela montre la puissance, en terme de réutilisabilité, de la programmation polytypique. Malheureusement, AG_{poly} n'est pas toujours la grammaire la plus appropriée pour écrire un algorithme.

Dans ce même ordre d'idée, et pour les raisons évoquées à la section 3.2 à propos des sommes, il existe dans `PolyP` une notion de correspondance entre constructeurs. Cela permet de spécifier, sans l'ambiguïté dûe aux constructeurs sum_l et sum_r, l'opération suivante :

$$\begin{array}{l} \varphi \stackrel{def}{=} \{c_1 \to d_1 \ldots c_n \to d_n\} \\ \varphi\ (c_k\ \overline{t}) = d_k\ (\varphi\ t_1) \ldots (\varphi\ t_n) \quad \forall k \end{array}$$

Cette opération est très proche de la correspondance étendue, que j'ai développée dans [1]. Ceci ne fait donc que confirmer l'utilité de cette notion.

4.5 Polymorphisme et Couplage(s)

La notion de couplage est également utile pour faire du polymorphisme. Prenons le type suivant :

$$\tau \ a = C_1 \texttt{of} \ a \mid C_2 \texttt{of} \ (\mathit{list} \ (\tau \ a))$$

En grammaire attribuée, ce type correspond à la grammaire :

$$\mathcal{G} = \left\{ \begin{array}{ll} C_1 : & \tau \leftarrow a \\ C_2 : & \tau \leftarrow l_\tau \\ cons_\tau : & l_\tau \leftarrow \tau * l_\tau \\ nil_\tau : & l_\tau \leftarrow \end{array} \right.$$

Cependant, l'intérêt d'écrire le type τ en utilisant le type *list* est d'identifier les arbres de $\mathcal{G}$ de type l_τ comme des listes. Ce qui se fait naturellement en spécifiant le couplage par correspondance suivant :

$$\begin{array}{l} Cor_{l_\tau \rightarrow list}(list) = \{l_\tau\} \\ Cor_{l_\tau \rightarrow list}(a) = \{\tau\} \end{array}$$

Pour cette raison, il n'a pas été nécessaire d'introduire la notion de composition de foncteurs dans AG_{poly}.

Conclusion

Devant la taille et la complexité sans cesse croissante des programmes informatiques, les diverses notions de généricité vont rapidement intervenir au premier plan.

Dans ce cadre, les grammaires attribuées offrent un formalisme simple et efficace, et qui possède quelques proprétés remarquables. Entre autres, citons sa capacité à modéliser avec un formalisme *unique* des programmes écrits pour des modèles d'évaluation *multiples*.

Remarquons que les grammaires attribuées ne sont pas seulement un formalisme abstrait représentant une sémantique, mais qu'elles sont utilisables en tant que language de programmation à part entière. Cependant, il faut souligner que chaque type de language se révèle particulièrement adapté à un certain type de calcul, et l'utilisation des grammaires attribuées ne doit pas occulter cette propriété.

En pratique, nous commençons à avoir une expérience de la traduction de languages fonctionnels en grammaire attribuées. Je compte entreprendre le même travail pour les languages impératifs et les languages à objets, pour lesquels nous avons déja des élements de départ. Parallèlement, l'analyse des méthodes de programmation générique propres à ces languages, comme je l'ai fait pour le polytypisme, permet de trouver de nouveaux moyens de spécifier les *couplages* entre algorithmes génériques et situations concrètes. Ainsi se trace une voie prometeuse pour obtenir de puissants outils de généricité.

Bibliographie

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

[2] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Draft, INRIA, July 1997. `ftp://ftp.inria.fr/INRIA/Projects/oscar/FNC-2/publications/SC97.ps.gz`.

[3] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

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

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

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

[7] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

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

[9] Martin Jourdan and Didier Parigot. Application development with the FNC-2 attribute grammar system. In Dieter Hammer, editor, *Compiler Compilers '90*, Lecture Notes in Computer Science, pages 85–94. Springer-Verlag, New York–Heidelberg–Berlin, October 1990 January 1992. Schwerin.

[10] Richard Kieburtz and Jeffrey Lewis. Algebraic design language. Technical report, Oregon Graduate Institute, 1994.

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

[12] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lect. Notes in Comp. Sci.*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

[13] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144, Cambridge, September 1991. Springer-Verlag.

[14] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *In Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997.

[15] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[16] Didier Parigot, Etienne Duris, Gilles Roussel, and Martin Jourdan. Les grammaires attribuées : un langage fonctionnel déclaratif. In *Journées Francophones des Langages Applicatifs*, pages 263–279, Val-Morin, Québec, January 1996. `ftp://ftp.inria.fr/INRIA/Projects/oscar/FNC-2/publications/jfla96.ps.gz`.

[17] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2881.ps.gz`.

[18] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[19] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press.

[20] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 306–313, La Jolla, CA, USA, 1995. ACM Press.