Schéma générique de développement par composition

Loïc Correnson, Etienne Duris,
Didier Parigot et Gilles Roussel

INRIA-Rocquencourt.
Gilles Roussel est à l'Université de Marne-la-Vallée.

Résumé

Depuis peu, la programmation générique suscite un intérêt grandissant dans différents paradigmes de programmation. Un principe souvent utilisé pour obtenir de la généricité est d'abstraire les calculs d'un programme par rapport à leur structure de données. Cette approche permet à ces spécifications génériques d'être instanciées pour un grand nombre de structures de données voisines. De plus, le programme peut ainsi être automatiquement adapté lorsque les structures de données évoluent.

La programmation polytypique, la programmation adaptive et les grammaires attribuées génériques sont des méthodes formelles de programmation génériques qui adoptent cette approche. La comparaison de ces méthodes nous a conduit à proposer un schéma commun de développement de programmes génériques. Cette méthode est basée sur deux concepts fondamentaux : la génération automatique de morphismes entre structures de données, et l'instanciation formelle des programmes génériques par composition, assistée par des outils de spécialisation.

1 Introduction

L'intérêt pour la programmation générique se développe dans plusieurs paradigmes de programmation. Quand un algorithme est spécifié sur une structure de données générale, la notion de généricité vient de la possibilité de réutiliser cet algorithme dans plusieurs contextes, c'est à dire sur différentes structures de données particulières. Bien que ce concept ne soit pas nouveau, la généricité est de plus en plus cruciale dans le développement et la maintenance de logiciels.

Il y a quelques années, Farrow *et al.* [7] ont développé un concept de généricité pour les grammaires attribuées [11, 18] qui a été récemment étendu [12, 1, 19]. Avec cette forme de généricité, il est possible d'instancier sur un type τ_1 n'importe quelle fonction f décrite par une grammaire attribuée sur un type τ_2. Le seul objet nécessaire, s'il existe, est un *couplage* décrit par une grammaire attribuée, i.e., un morphisme qui transforme chaque structure de type τ_1 en une structure de type τ_2. Le processus d'instanciation est alors obtenu par composition de cette fonction de couplage avec la fonction f. Dans ce contexte, le processus nécessaire de spécialisation consiste à appliquer une transformation

de programme qui élimine les constructions des structures de données intermédiaires dues à la composition. Pour les grammaires attribuées, ce problème de spécialisation à été résolu par Ganzinger et Giegerich grâce à leur algorithme de *composition descriptionnelle* [8]. Cette transformation évite la construction de la représentation intermédiaire de type τ_2 lors de la composition, et la fonction originelle f se trouve donc spécialisée pour le type τ_1.

Ce problème d'élimination des structures intermédiaires, souvent appelé *déforestation* [21], a aussi été étudié dans le domaine de la programmation fonctionnelle [16, 20, 17]. Les comparaisons [6, 4] entre les méthodes de déforestation en programmation fonctionnelle et en grammaires attribuées ont conduit à une extension de la composition descriptionnelle appelée *composition symbolique* [5]. Cette nouvelle méthode transforme efficacement des programmes fonctionnels pour lesquels les méthodes fonctionnelles connues sont impuissantes. Par ailleurs, l'importance de la composition symbolique dans les mécanismes de généricité des grammaires attribuées nous a conduit à les comparer avec des méthodes fonctionnelles de programmation générique.

En programmation fonctionnelle, le polymorphisme fournit une forme de généricité. Cependant, une fonction polymorphe ne peut être réutilisée que pour des types qui ont exactement la même structure de base (listes, arbres...). Afin de relâcher cette contrainte, la notion de *programmation polytypique* [10, 15] a été introduite. Une fonction polytypique est une fonction définie par induction sur la structure d'un type défini par l'utilisateur [9]. Elle est donc instanciable sur n'importe quelle structure *via* la génération automatique de morphismes.

En programmation orientée objet, il existe une autre approche de la généricité dirigée par la structure, provenant de la notion de *programmation adaptive* [13] et appelée *programmation tree-traversal* [14]. Cette approche permet de générer automatiquement un programme qui "traverse" une structure de données quelconque en ne s'intéressant qu'à ses feuilles. Cette technique peut être vue comme une génération de morphismes entre la structure de données d'entrée et la liste de ses composants.

Le but de cet article est de présenter un schéma général de développement, intégrant différentes techniques de programmation génériques. L'unification de ces méthodes est basée sur les concepts suivants :

La programmation générique est obtenue par la composition de fonctions. Ces fonctions sont de deux sortes : des fonctions classiques qui implantent des algorithmes génériques sur une structure de donnée appropriée, et des fonctions de morphismes qui implantent des traductions entre structures de données voisines.

La spécification des morphismes fournit la véritable généricité. La description du passage d'une structure à une autre donne le moyen d'effectuer l'instanciation d'un algorithme par composition. La plupart des morphismes impliqués dans ces compositions peuvent être dérivés automatiquement, à l'aide d'un petit méta-langage. Ce dernier peut avantageusement être indépendant (ne pas inclure) le langage original. Néanmoins, ces morphismes peuvent également être écrits à la main.

Le processus de spécialisation réalise l'instanciation effective. L'implantation des compositions doit être dédiée et adaptée à son contexte par transformation. Plutôt que d'appliquer une méthode *ad-hoc* pour chaque système générique particulier, il est préférable d'utiliser une méthode *générale* de transformation (déforestation, fusion) qui effectue en même temps la composition, l'élimination des structures intermédiaires et de l'évaluation partielle.

La section 2 présente brièvement les méthodes de programmation générique considérées : la programmation polytypique, la programmation *tree-traversal* et les grammaires attribuées génériques. La section 3 montre comment les deux premières méthodes peuvent être reformulées avec l'approche compositionnelle des grammaires attribuées. Avant de conclure, la section 4 décrit les avantages et les perspectives d'une telle approche pour le développement de programmes génériques.

Remarques préliminaires : afin de mettre en évidence les fertilisations croisées entre ces différentes méthodes de généricité, cet article présente plutôt leurs principes généraux que leurs aspects techniques. Les trois méthodes ne sont pas présentées intégralement et les notations sont incomplètes. Aussi, nous présentons nos excuses à leurs auteurs pour les importantes simplifications qui sont faites dans les présentations qui suivent.

2 Trois méthodes de programmation générique

2.1 La programmation polytypique

De façon classique en programmation fonctionnelle, les fonctions ne peuvent être spécifiées que pour un seul type polymorphe. Cependant, la plupart des fonctions peuvent être abstraites par rapport à leur type. Par exemple, le nombre de feuilles d'un arbre et la longueur d'une liste sont des fonctions semblables. La programmation polytypique propose une façon de spécifier de telles fonctions abstraites, afin de pouvoir les instancier sur n'importe quel type. Elle est basée sur le résultat théorique énonçant que n'importe quel type polymorphe peut être exprimé par une combinaison de sommes, produits et éléments de plusieurs sortes. Cette propriété est fortement liée aux notions catégorielles d'algèbre initiale et de foncteur [16, 9].

Considérons par exemple les types polymorphes *list* α et *tree* α :

$$\begin{aligned} &\mathit{list}\ \alpha = \mathit{nil} \mid \mathit{cons}\ (\alpha\ ,\ \mathit{list}\ \alpha) \\ &\mathit{tree}\ \alpha = \mathit{leaf}\ (\alpha) \mid \mathit{node}\ (\mathit{tree}\ \alpha\ ,\ \mathit{tree}\ \alpha) \end{aligned}$$

Ces types peuvent être abstraits automatiquement par leurs foncteurs :

$$\begin{aligned} \mathcal{F}_{\mathit{list}} &= \mathtt{Empty} + (\mathtt{Par} \times \mathtt{Rec}) \\ \mathcal{F}_{\mathit{tree}} &= \mathtt{Par} + (\mathtt{Rec} \times \mathtt{Rec}) \end{aligned}$$

où `Rec` représente le type récursif et `Par` le paramètre polymorphe α.

Plus généralement, n'importe quel terme peut être abstrait de la même manière par les constructeurs suivants :

$+\ c\ x$	pour l'alternative $(c\ x)$ d'une somme (e.g. l'alternative $cons(h,t)$ ou nil d'une liste).
$\times\ x\ x'$	pour le produit (x, x').
`Par` x	pour les paramètres intéressants.
`Rec` x	pour les composants récursifs.
`Const` x	pour les paramètres cachés.
`Empty`	pour les valeurs vides.

Par exemple, la liste $[a; b; c]$ est abstraite en $+$ $(cons)$ $(a\ ,\ [b; c])$, puisqu'il s'agit d'une alternative *cons* d'une liste, avec a comme élément de tête et $[b; c]$ comme sous liste (de queue).

Écrire un programme générique

Nous allons spécifier la fonction polytypique *length* à partir des constructeurs énumérés ci-dessus. Intuitivement, puisque une somme $(+)$ représente des alternatives possibles, la longueur d'une somme est la longueur de l'alternative considérée. De même, calculer la longueur d'un produit $(\times)$ de termes consiste à ajouter leurs longueurs. Finalement, la longueur d'un paramètre est 1 si on doit le prendre en compte et 0 sinon. Ces remarques intuitives aboutissent à la fonction polytypique *length* présentée ci-après (*add* représente l'addition entière) :

```
polytypic length x =
  match (out x) with
   + c y      → (length y)
   × y y'     → (add (length y) (length y'))
   Rec y      → (length y)
   Par y      → 1
   Const y    → 0
   Empty      → 0
```

Spécification d'une instance

Remarquons que ce programme nécessite un opérateur permettant d'abstraire une structure d'entrée en terme de sommes et de produits. Ceci est pris en charge par l'opérateur `out` du langage polytypique. C'est un morphisme initial qui doit être défini pour chaque type possible de son argument. En programmation polytypique [9], il existe un *type-checker* qui détecte tous ces types et définit automatiquement l'opérateur `out`.

Processus d'instanciation et de spécialisation

Considérons l'implantation de la fonction polytypique *length* pour un argument de type *tree* α : l'expression *length* $(x$: *tree* $\alpha)$ doit être dédiée en $length_{tree\,\alpha}\ x$. L'opérateur `out` doit alors être défini pour le type *tree* α comme suit :

$$
\begin{array}{ll}
\texttt{out}\ (\mathit{leaf}\ n) = +\ (\mathit{leaf})\ (n) & \\
\texttt{out}\ n = \texttt{Par}\ n & \text{si } n \text{ est de type } \alpha \\
\texttt{out}\ (\mathit{node}\ x\ y) = +\ (\mathit{node})\ (x, y) & \\
\texttt{out}\ (x, y) = \times\ x\ y & \\
\texttt{out}\ x = \texttt{Rec}\ x & \text{si } x \text{ est de type } \mathit{tree}\ \alpha
\end{array}
$$

Pour spécialiser cette instanciation, tous les constructeurs abstraits ($+$, $\times$, `Par`...) doivent être éliminés. Dans [9], c'est une méthode d'évaluation partielle *ad-hoc* [1], dédiée aux constructeurs introduits par l'opérateur `out`, qui se charge de cette phase. La fonction spécialisée obtenue est la suivante :

$$
\begin{array}{l}
\texttt{let}\ \mathit{length}_{\mathit{tree}\alpha}\ x = \\
\quad \texttt{match}\ x\ \texttt{with} \\
\quad\quad \mathit{node}\ a\ b \rightarrow (\mathit{add}\ (\mathit{length}_{\mathit{tree}\alpha}\ a)\ (\mathit{length}_{\mathit{tree}\alpha}\ b)) \\
\quad\quad \mathit{leaf}\ n \rightarrow 1
\end{array}
$$

Pour une présentation plus complète de la programmation polytypique, nous invitons le lecteur à se reporter à [9].

2.2 La programmation *tree-traversal*

Le concept de *programmation adaptive* [13] étend le paradigme classique de programmation orientée objet. Il autorise plus de flexibilité entre les fonctions (méthodes) et les types de données (classes) en les liant par des spécifications de navigation. Cette section traite d'une implantation de ce concept appelée programmation *tree-traversal* [14].

Écrire un programme générique

Un exemple intéressant de ce type de généricité par programmation *tree-traversal* est le problème suivant : on désire surligner un mot particulier w dans un texte. Ce problème peut intuitivement être spécifié comme ceci :

```
at word {
    if (this.val=w) then this.highlight();
}
```

où 'this' dénote le mot courant, w est le mot à surligner et 'highlight()' est une méthode qui surligne l'objet auquel elle est appliquée. Cependant, si le document continent des paragraphes, qui peuvent eux-mêmes contenir des lignes contenant des mots, la spécification ci-dessus est incomplète.

Spécification d'instance

Pour surligner le mot w, le programmeur doit écrire à la main de nombreuses méthodes de recherche permettant d'atteindre le mot à l'intérieur de la struc-

1. Par *ad-hoc*, nous voulons dire qu'aucune *véritable* méthode de déforestation n'est exhibée.

ture du document. La technique *tree-traversal* utilise l'instruction `traverse` qui permet de générer automatiquement ces méthodes.

```
traverse
    from document to word
        at word {
            if (this.val=w) then this.highlight();
        }
```

Processus d'instanciation

Pour une telle spécification d'instanciation, le système produit une méthode 'search' pour chaque classe qui contient (même indirectement) un mot. Dans notre exemple, les méthodes produites sont les suivantes :

```
document::search(w) {
    for paragraph in this.paragraphs()
        { paragraph.search(w); }
}
paragraph::search(w) {
    for line in this.lines()
        { line.search (w); }
}
line::search(w) {
    for word in this.words()
        { word.search(w); }
}
word::search(w) {
    if (this.val=w) then this.highlight();
}
```

Le système *tree-traversal* génère automatiquement de telles méthodes, et évite de les écrire à la main. Cependant, cette approche est bien plus utile lorsque la structure de données doit évoluer. Par exemple, supposons que la structure de document doive être modifiée pour accepter des 'trucs' qui sont des paragraphes *ou* des tables (pouvant contenir des mots). Les méthodes de recherche seront alors automatiquement mises à jour, même si la spécification du problème reste inchangée. En fait, plus un programme contient de spécifications *tree-traversal*, plus il est tolérant à l'évolution des structures de données.

Pour une présentation plus complète du système *tree-traversal*, le lecteur est invité à se référer à [14].

2.3 Les grammaires attribuées génériques

Les grammaires attribuées [11, 18] sont des spécifications déclaratives dirigées par la structure des données. Elles permettent de spécifier, sur chaque constructeur de type, *ce* qui doit être calculé plutôt que *comment* ce doit être calculé. Les programmes sont spécifiés par des équations orientées, guidées par la structure des données.

Par exemple, pour calculer la longueur d'une liste, l'idée est de définir un attribut (i.e., une valeur) *length* pour chaque liste x. Une telle occurrence d'attribut est dénotée par $x.length$. Comme chaque liste est construite soit par un *nil* soit par un *cons*, la grammaire attribuée spécifiant ce calcul est la suivante :

$$
\begin{array}{l}
cons\ h\ t \rightarrow \\
\quad this.length = (add\ t.length\ 1) \\
nil \rightarrow \\
\quad this.length = 0
\end{array}
$$

où *this* représente la liste considérée.

Prenons un autre exemple : un document est constitué d'une liste de chapitres, chacun d'entre eux étant une liste de sections, etc. Le but est alors de calculer la liste des mots du document (pour les compter, ou pour rechercher un mot particulier...).

Écrire une grammaire attribuée générique

Un type adapté à l'abstraction d'une structure contenant des éléments est le type *bag* suivant :

$$bag\ element = union\ (bag, bag)\ \mid single\ (element)\ \mid empty$$

La liste des éléments d'un tel ensemble est construite par accumulation. La grammaire attribuée *bag.list* spécifiant cette construction nécessite donc deux attributs, *outlist* et *inlist* (ce dernier est initialisé avec la valeur *nil*) :

$$
\begin{array}{l}
single\ a \rightarrow \\
\quad this.outlist = (cons\ a\ this.inlist) \\
empty \rightarrow \\
\quad this.outlist = this.inlist \\
union\ a\ b \rightarrow \\
\quad this.outlist = b.outlist \\
\quad b.inlist = a.outlist \\
\quad a.inlist = this.inlist
\end{array}
$$

Spécification d'instance

Supposons que la structure complète du document soit définie à l'aide de différents sous-types et de leurs constructeurs :

$$
\begin{array}{l}
document = doc\ (chapter, document)\ \mid end\_doc \\
chapter = chap\ (section, chapter)\ \mid end\_chap \\
section = sect\ (paragraph, section)\ \mid end\_sect \\
paragraph = par\ (word, paragraph)\ \mid end\_par
\end{array}
$$

Afin d'instancier la construction de la liste de mots pour ces documents, l'idée est de générer une fonction de couplage qui transforme un document en une structure *bag*.

Soit $\mathcal{C}or$ la propriété (ou relation) qui associe chaque *document*, *chapter*, *section*, et *paragraph* à une structure de type *bag*, et chaque mot *word* à un *element*. Dans ce cas, $\mathcal{C}or$ définit complètement le morphisme entre *bag* et la structure du document. Cette relation de correspondance $\mathcal{C}or$ peut être spécifiée comme ceci :

$$\begin{aligned} \mathcal{C}or(bag) &= \{document, chapter, section, paragraph\} \\ \mathcal{C}or(element) &= \{word\} \end{aligned}$$

Instanciation

Il est possible d'écrire à la main une grammaire attribuée qui implante un morphisme vérifiant la relation $\mathcal{C}or$: pour chaque document, chapitre, section et paragraphe, la grammaire attribuée *document.coupling* suivante définit la structure *bag* équivalente.

$$\begin{aligned} &doc\ c\ d \rightarrow \\ &\quad this.coupling = (union\ c.coupling\ d.coupling) \\ &chap\ s\ c \rightarrow \\ &\quad this.coupling = (union\ s.coupling\ c.coupling) \\ &sect\ p\ s \rightarrow \\ &\quad this.coupling = (union\ p.coupling\ s.coupling) \\ &par\ w\ p \rightarrow \\ &\quad this.coupling = (union\ (single\ w)\ p.coupling) \end{aligned}$$

Plutôt que d'écrire cette grammaire attribuée de couplage à la main, il existe des algorithmes qui peuvent l'inférer automatiquement. Les notions de relation de correspondance et de grammaire de couplage sont formalisées dans [12, 2]. Les conditions permettant à ce couplage d'être bien défini y sont étudiées. Plus précisément, étant donnés une grammaire de sortie (e.g., *bag*), une grammaire d'entrée (e.g., *document*) et un couplage valide entre elles (e.g., $\mathcal{C}or$), il existe un algorithme qui, sous certaines conditions, construit la grammaire attribuée de couplage (e.g., *document.coupling*).

Soit Γ l'environnement de programmation contenant la description des types *bag* et *document*. Soit $\mathcal{C}$ l'algorithme qui infère la grammaire attribuée de couplage *document.coupling*. L'instanciation peut alors être formalisée comme ceci :

$$\mathcal{C}or, \Gamma \stackrel{\mathcal{C}}{\Rightarrow} document.coupling$$

Processus de spécialisation par déforestation

Il est maintenant facile de composer la grammaire attribuée *document.coupling* avec la grammaire attribuée *bag.list*. À partir de ces deux grammaires attribuées, la composition symbolique [5] produit une nouvelle grammaire attribuée *document.list* suivante, qui ne construit plus la structure *bag* intermédiaire :

$$
\begin{array}{l}
doc\ c\ d \rightarrow \\
\quad this.outlist = d.outlist \\
\quad d.inlist = c.outlist \\
\quad c.inlist = this.inlist \\
chap\ s\ c \rightarrow \\
\quad \ldots \\
sect\ p\ s \rightarrow \\
\quad \ldots \\
par\ w\ p \\
\quad this.outlist = (cons\ w\ p.inlist)
\end{array}
$$

Si SC représente la composition symbolique, le processus de spécialisation est défini par :

$$document.coupling \circ bag.list \stackrel{\mathrm{SC}}{\Rightarrow} document.list$$

Il est important de noter ici que la génération du morphisme et l'application de la déforestation sont complètement séparées l'une de l'autre. Le lecteur pourra trouver plus de détails sur la composition symbolique dans [5].

3 L'approche compositionnelle

Cette section présente une reformulation des méthodes polytypique et *tree-traversal*.

3.1 La méthode polytypique

Dans l'approche polytypique classique, des constructeurs spécifiques ($+$, $\times$, `Rec`, `Par`...) sont introduits pour manipuler à la fois les données et leur structure. Définissons maintenant le type suivant :

$$
\begin{array}{rcl}
\mathcal{F} & = & Sum\ (Object, \mathcal{F}) \\
& | & Prod\ (\mathcal{F}, \mathcal{F}) \\
& | & Par\ (Object) \\
& | & Rec\ (\mathcal{F}) \\
& | & Obj\ (Object)
\end{array}
$$

Le type $\mathcal{F}$ permet de spécifier la fonction *length* dans le langage original :

$$
\begin{array}{l}
\texttt{let}\ length\ x = \texttt{match}\ x\ \texttt{with} \\
\quad \begin{array}{ll}
Sum\ c\ y & \rightarrow (length\ y) \\
Prod\ y\ y' & \rightarrow (add\ (length\ y)\ (length\ y')) \\
Rec\ y & \rightarrow (length\ y) \\
Par\ y & \rightarrow 1 \\
Obj\ z & \rightarrow 0
\end{array}
\end{array}
$$

Supposons maintenant que l'on ait besoin de la fonction *length* sur le type *tree* α. L'instanciation correspondante est effectuée par un morphisme explicite

depuis le type *tree* α vers le type $\mathcal{F}$. Ce dernier s'écrit sous la forme de la fonction *out_tree*α suivante :

$$\begin{array}{l}\texttt{let } out\_tree\alpha\ t = \texttt{match } x \texttt{ with} \\ \quad node\ a\ b \ \ \rightarrow (Sum\ (node)\ (out\_tree\alpha\ a)\ (out\_tree\alpha\ b)) \\ \quad leaf\ y \ \ \rightarrow (Sum\ (leaf)\ (Par\ y))\end{array}$$

Il est important de noter que *out_tree*α n'est plus un opérateur comme `out` l'était dans le langage polytypique classique, mais une simple fonction écrite dans le langage fonctionnel. La fonction *length* sur le type *tree* α est donc obtenue par la composition suivante :

$$length\_tree\alpha\ t = (length\ (out\_tree\alpha\ t))$$

La déforestation appliquée sur cette composition élimine la construction des termes de type $\mathcal{F}$, et produit la fonction *length* attendue pour le type *tree* α.

De la même façon que l'opérateur polytypique `out` a pu être inféré, cette fonction *out_tree*α peut être générée automatiquement.

Reformulation du système

En considérant l'exemple précédent, le système polytypique peut être reformulé comme suit :

$$\frac{\tau \overset{\mathcal{C}}{\Rightarrow} out\_\tau}{f, \tau \Rightarrow f \circ out\_\tau} \qquad f \circ out\_\tau \overset{deforestation}{\Rightarrow} f\_\tau$$

où $\circ$ est la composition standard dans le langage originel. Une méthode de déforestation générale peut alors être appliquée plutôt qu'une méthode restreinte aux fonctions polytypiques. Par exemple, le système HYLO [17], ou la composition symbolique [5], peuvent avantageusement prendre en charge le processus de spécialisation dans l'approche polytypique. Une comparaison plus détaillée entre les grammaires attribuées génériques et la programmation polytypique est développée dans [3].

3.2 La méthode *tree-traversal*

Approche compositionnelle

Reprenons l'exemple présenté dans la section 2.2, et considérons le document D suivant :

$$D \rightarrow \begin{cases} P_1 \rightarrow \{L_1 \rightarrow \{w_1\,;w_2\}\} \\ P_2 \rightarrow \{L_2 \rightarrow \{w_3\,;w_4\}\,; L_3 \rightarrow w_5\} \end{cases}$$

D contient deux paragraphes P_1 et P_2. Le paragraphe P_1 contient une ligne L_1 qui contient deux mots w_1 et w_2, etc. Les méthodes 'search' permettent de parcourir le document D dans l'ordre suivant :

$$D\,;P_1\,;L_1\,;w_1\,;w_2\,;P_2\,;L_2\,;w_3\,;w_4\,;L_3\,;w_5$$

Considérons le type T d'une telle liste hétérogène. La méthode *tree-traversal* peut alors être décomposée en deux programmes. Étant donné un document, le premier programme produit la liste des mots du document (liste de type T). Le deuxième programme surligne les mots w d'une liste de type T.

Maintenant, la véritable spécialisation peut être effectuée sur cette composition et ainsi la construction de la liste (de type T) sera éliminée. A notre connaissance, il n'existe pas en programmation orientée objet de telle méthode de spécialisation.

4 Une généricité compositionnelle

Cette section présente un système de programmation générique abstrait, i.e. défini indépendamment de n'importe quel langage. Il est basé sur deux processus bien séparés : un processus d'instanciation et d'un processus de spécialisation. Le premier est un algorithme de couplage qui produit un morphisme entre deux structures de données. Ce morphisme est implanté par une fonction qui traduit un terme en une représentation intermédiaire ; cette fonction pourra donc être composée avec n'importe quel programme travaillant sur cette représentation intermédiaire. Le deuxième processus applique une méthode de déforestation afin de spécialiser de telles compositions. Étant donnés une spécification de morphisme M, l'algorithme de couplage $\mathcal{C}$, une fonction ou un programme générique f, le morphisme m produit par $\mathcal{C}$, le programme g spécialisé par une méthode de déforestation et un environnement de programmation Γ, le système de généricité est défini comme suit :

$$\frac{M, \Gamma \overset{\mathcal{C}}{\Rightarrow} m}{f, M, \Gamma \Rightarrow f \circ m} \qquad \frac{f \circ m \overset{deforestation}{\Rightarrow} g}{f \circ m \Rightarrow g}$$

Le processus de spécialisation de la méthode polytypique est très proche de celui utilisé dans la méthode de généricité pour les grammaires attribuées. Dans une première approche, ils peuvent être apparentés à une méthode de déforestation. Cependant, pour la méthode polytypique, le processus de spécialisation utilise une forme qui est plus proche d'une évaluation partielle que d'une véritable méthode de déforestation. En nous référant aux comparaisons [4, 5] entre les méthodes de déforestation en programmation fonctionnelle et en grammaires attribuées, il nous semble plus intéressant d'utiliser un algorithme général de déforestation plutôt qu'une méthode *ad-hoc*. En effet, dans cette approche compositionnelle, la programmation générique bénéficie gratuitement de toute la puissance des méthodes de transformation des grammaires attribuées.

Par contre, pour la programmation orientée objet, la définition d'un tel processus de spécialisation est encore un problème ouvert. Dans tous les cas, il est important d'identifier séparément le processus d'instanciation du processus de spécialisation.

De plus, pour que n'importe quelle méthode générale de déforestation puisse être utilisée gratuitement, il faut que le programme générique et le morphisme soient écrits dans le même langage. Il s'avère donc plus intéressant de spécifier

les morphismes à l'aide de méta-langages (relation de correspondance) complètement indépendant du langage d'origine. De plus, cette séparation et cette indépendance vis-à-vis du langage d'origine permet de transposer plus facilement les techniques de génération de morphismes d'un paradigme de programmation vers un autre. Par exemple, la génération automatique de l'opérateur `out` (de la méthode polytypique) nous a permis de définir une méthode semblable de génération de couplage de grammaires attribuées. De la même manière, le concept de relation de correspondance laisse entrevoir une nouvelle technique de génération de l'opérateur `out`.

Il serait donc intéressant d'unifier, dans un même formalisme, l'ensemble de ces méta-langages de spécification de morphismes, dans le but de faciliter les comparaisons et la réutilisation de ces techniques dans différents paradigmes de programmation.

5 Conclusion

Dans cet article, nous exhibons un concept général de programmation générique unifiant différentes méthodes existantes. Plutôt que de les opposer, ce schéma général les rapproche par des fertilisations croisées. Chaque méthode possède des avantages spécifiques, mais également des limitations, en offrant des types de généricités différents et complémentaires. Afin de faciliter leurs comparaisons et de comprendre leurs différences, il nous a semblé intéressant de séparer la génération des morphismes, des processus d'instanciation et de spécialisation de code.

La composition symbolique – comme méthode de déforestation – est l'outil de base qui permet de spécialiser un algorithme sur de nouvelles structures par composition avec des morphismes. Dès qu'une méthode de déforestation est connue et identifiée pour un paradigme de programmation donné, il est possible de développer rapidement et facilement une méthode efficace de généricité.

En outre, d'autres méthodes génériques de programmation et de spécialisation pourraient être incluses dans le même schéma général. De notre point de vue, les futurs travaux de recherche devront porter sur la possibilité d'unifier les différentes méthodes de spécification de morphismes et rechercher des méthodes automatiques ou semi-automatiques pour les générer.

Références

[1] John Boyland and Susan L. Graham. Composing tree attributions. In *21st ACM Symp. on Principles of Programming Languages*, pages 375–388, Portland, Oregon, January 1994. ACM Press.

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

[3] Loïc Correnson. Programmation polytypique avec les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1997.

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

[5] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1998.

[6] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996.

[7] 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.

[8] 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).

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

[10] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996.

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

[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] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[14] Cristina Videira Lopes and Karl J. Lieberherr. AP/S++: A CASE-study of a MOP for purposes of software evolution. Technical Report NU-CCS-95-?, Xerox PARC and Northeastern University, November 1995.

[15] Lambert Meertens. Calculate polytypically. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 1–16, Aachen, September 1996. Springer-Verlag.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science*, volume 73, pages 231–248, 1990. (Special issue of selected papers from 2'nd European Symposium on Programming).