Rapport de Stage

Généricité dans les Grammaires Attribuées

Loïc Correnson

1^er^ juillet 1996

Introduction

Les grammaires attribuées ont été introduites par Knuth [Knu68] pour décrire les programmes dirigés par la syntaxe. Ce paradigme de programmation constitue un outil particulièrement adapté aux applications de traitement des languages, tels que les compilateurs et les traducteurs.

Une grammaire attribuée est assimilable à un algorithme sur une certaine structure. On peut souhaiter réutiliser cet algorithme sur d'autres structures "comparables". Ainsi, on disposerait d'une forme de généricité.

Dans un article récent [BJPR93] est exposé un mécanisme de couplage de deux grammaires par une grammaire attribuée. Partant d'une correspondance entre les terminaux et non-terminaux des deux grammaires G_1 et G_2, on peut générer une grammaire attribuée qui extrait d'un arbre de G_1 l'arbre de G_2 qui lui correspond. Les algorithmes qui s'appliquent sur G_2 peuvent alors s'appliquer sur G_1, en effectuant une composition symbolique avec la grammaire attribuée de couplage. L'intérêt de cette composition symbolique est que l'arbre sur G_2 ne sera, en fait, pas construit. On obtiendra alors une véritable instanciation sur G_1 de l'algorithme initialement défini sur G_2.

J'ai complété l'article en question qui ne donnait aucun algorithme d'exploration de différentes solutions partielles. Puis, cet algorithme de couplage a été implanté dans le système Fnc-2 [JP91], qui possède un opérateur de méta-composition particulièrement efficace [Rou94].

Ensuite, j'ai cherché de quelle mainère il faut écrire un gène pour qu'il puisse être instancié sur le plus de cas possibles. J'ai ainsi définis une forme standard d'une grammaire attribuée ainsi qu'un algorithme de réduction pour s'y ramener.

Abstract

Knuth introduced attributes grammars to study syntax-oriented programs [Knu68]. This paradigm is well adapted to langage features, such as compilation or translation.

An attributed grammar is an algorithm defined on a particular structure. We want to apply this algorithm on a similar structure, as if this algorithm was an generic one.

A mechanism for coupling two grammars by an attributed grammar is described in [BJPR93]. With a correspondance between the terminals and non-terminals of two grammars G_1 and G_2, it is possible to generate a coupling attributed grammar wich extracts from tree en G_1 a tree on G_2. Then, the genericity comes with a descriptional composition, wich make an instanciation on G_1 of an algorithm on G_2.

In this paper, we extend [BJPR93]. We have implemented the coupling algorithm in Fnc-2 system, and then we try to answer the question : "How to write a gene such that it will be instanciable on a large number of grammars ?". As a good solution, we have defined a standard form for an attributed grammar, and a reduction alorithm.

Table des matières

Chapitre 1

Grammaires Attribuées

1.1 Définitions

Pour se fixer les idées, rappelons que les grammaires hors-contexte furent introduites afin de formaliser des techniques d'analyse de fichiers textes ayant une certaine structure, comme le texte source d'un programme [Knu68].

Par la suite, avec les grammaires attribuées et leur arsenal de générateurs d'évaluateurs [DJL88], ce paradigme de programmation s'est naturellement affranchi des textes sources. En effet, toute la théorie développée autour des grammaires attribuées est utilisable dès que l'on veut faire un calcul sur une structure d'arbre.

Ainsi, le système Fnc-2 [JP91] ne travaille que sur des arbres abstraits, mais une partie frontale permet d'extraire d'un texte source un arbre de syntaxe abstraite qui lui correspond.

Grammaires & Arbres

Une grammaire hors-contexte est un quadruplet $G = (T, N, Z, P)$.

T est un ensemble de symboles terminaux, qui représentent les élements syntaxiques qui peuvent composer le texte source.

N est un ensemble de symboles non-terminaux, qui représentent des regroupements de terminaux et non-terminaux ayant une certaine valeur syntaxique.

Z est le non-terminal racine, représentant la totalité du texte source.

P est un ensemble de productions. Une production indique comment des terminaux et des non-terminaux se dérivent en d'autres non-terminaux. On note une production ainsi :

$p : X_0 \rightarrow X_1 \cdots X_n$ avec $X_0 \in N$ et $X_i \in (N \cup T)$

Un arbre de syntaxe abstraite, ou arbre de dérivation, d'une grammaire G est un arbre (fini) dont les nœuds sont étiquetés par un symbole dans $(N \cup T)$, et par une production dans P, et qui vérifie:

- la racine est étiquetée par Z.

- les feuilles sont étiquetées par des terminaux éléments de T.

- les nœuds internes sont étiquetées par des non-terminaux éléments de N, et par une production de P.
- si un nœud u est étiqueté par X_0 et p, alors p doit être de la forme :

  $$p\ : X_0 \rightarrow X_1 \cdots X_n$$

  où $X_1 \cdots X_n$ sont les fils de u.

On parle aussi de sous-arbres, la racine pouvant alors être un élément quelconque de N. Dans la suite, on notera souvent t un tel arbre, u un nœud interne, $label(u)$ et $prod(u)$ le non-terminal et la production au nœud u.

Grammaires Attribuées

Une grammaire attribuée (GA) est un triplet (G, A, F).

$G = (N, T, Z, P)$ est une grammaire hors contexte (cf. la définition ci-dessus), et $A = \bigcup A(X)$ est un ensemble d'attributs pour chaque symbole X de $(N \cup T)$. Les attributs d'un symbole doivent se scinder en deux sous-ensembles disjoints, AS et AH, appelés respectivement attributs *synthétisés* et *hérités*. Enfin, $F = \bigcup F(p)$ est un ensemble de règles sémantiques pour chaque production p.

Ces règles sémantiques permettent en chaque nœud u de calculer une partie des attributs de $label(u)$ ou de ses fils. Si $prod(u) = X_0 \rightarrow X_1 \cdots X_n$ alors la règle permettant de calculer un attribut a peut prendre deux formes :

- soit $a\ := b$
- soit $a\ := f(a_1, \cdots, a_n)$

avec $a \in AS(X_0) \cup AH(X_1, \cdots, X_n)$ et $b, a_1, \cdots, a_n \in AH(X_0) \cup AS(X_1, \cdots, X_n)$.

Pour chaque attribut, il ne doit pas exister de définition multiple. Donc chaque attribut de $AS(X_0) \cup AH(X_1, \cdots, X_n)$ doit être défini exactement une fois.

Si l'on a un arbre abstrait sur G, on peut le décorer, en ajoutant à chaque nœud u des étiquettes avec les attributs de $label(u)$. On obtient ainsi ce que l'on appelle un arbre attribué.

1.2 Evaluateurs

Pour un arbre attribué donné, quand on collecte toutes les règles sémantiques de chaque noeud de l'arbre, on obtient un système d'équations portant sur l'ensemble des attributs de l'arbre. Résoudre ce système peut bien sûr être problématique, surtout si des circularités apparaissent. Bon nombre des travaux sur les grammaires attribuées portent sur l'art et la manière de générer automatiquement des évaluateurs qui résolvent efficacement ces systèmes d'équations. En fait, le problème est très difficile dans sa version la plus générale, mais pour une classe suffisamment large de grammaires, on sait construire un évaluateur.

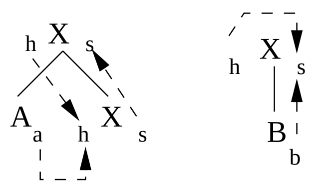

FIG. 1.1 – *Graphes de dépendance locaux*

GA non-circulaires

Pour générer un évaluateur d'attributs, le problème principal reste la circularité. En effet, si le système à résoudre contient une équation de la forme $a = f(a)$ ou similaire, il devient impossible de trouver une solution, sauf propriétés particulières de f, invérifiables en général.

Cette situation est facile à détecter sur un arbre attribué. Etant donné les règles sémantiques, on peut calculer un graphe de dépendance entre les attributs. Si la fermeture transitive de ce graphe contient une relation $a \to a$, le graphe est circulaire et la résolution est impossible.

Cette vérification a un inconvénient majeur par son caractère dynamique (on refait les calculs pour chaque arbre). On cherche plutôt une vérification statique permettant de savoir si une GA risque de produire des circularités. Pour y parvenir, on calcule un graphe de dépendance local à chaque production. Puis, on effectue une sorte de fermeture de tout ces graphes, en tenant compte de tous les arbres attribués possibles. Ces graphes ne reflètent pas les dépendances entre les occurrences d'attributs, mais entre les "types" d'attributs. Prenons un exemple :

Soit une GA dans laquelle on a :

$$p \ : \left\{ \begin{array}{l} X \to AX_1 \\ \quad X_1.h \ := f(A.a, X.h) \\ \quad X.s \ := X_1.s \end{array} \right.$$

$$q \ : \left\{ \begin{array}{l} X \to B \\ \quad X.s \ := g(B.b, X.h) \end{array} \right.$$

où s, a et b sont des attributs synthétisés respectifs de X, A et B, et h est un attribut hérité de X.

On est dans une situation où il y a un nombre infini d'arbres possibles, X dénotant une suite de A terminée par un B. Pourtant, si la donnée de p n'indique aucune dépendance locale entre $X.s$ et $X.h$, on voit bien que q introduit dans tous les arbres une dépendance entre ces deux attributs. Donc, s dépend de h (Cf figure 1.1).

On voit sur cet exemple que l'on peut calculer en un temps fini le graphe des dépendances sur les"types" d'attributs. Par ailleurs des travaux ont permis d'optimiser grandement ces tests de circularité [JP88].

On appelle grammaire attribuée non-circulaire une GA dont le graphe défini ci-dessus ne comporte aucun cycle. Cette classe de GA couvre la plupart des cas pratiques que l'on peut rencontrer en programmation dirigée par la syntaxe, de même que les grammaires LR(k) couvrent la presque totalité des grammaires hors-contexte utilisées. Qui plus est, on peut montrer que pour toute GA fortement non-circulaire, on peut construire un évaluateur (algorithme) qui calcule la valeur de tous les attributs, pour tout arbre attribué sur cette GA.

Stratégie d'évaluation

Pour calculer la valeur des attributs, de multiples solutions sont possibles. On peut cependant donner une idée des principes utilisés.

Si à un nœud u on a une règle sémantique $a := f(a_1, \ldots, a_n)$, et si tous les a_i sont déjà calculés, alors on peut calculer a. Sinon, il faut examiner les autres noeuds de l'arbre, jusqu'à ce que tous les a_i soient disponibles. On remarque que l'ordre dans lequel les attributs sont évalués devient vite très déterminant en termes d'efficacité.

Une méthode directe pour déterminer cet ordre est de faire un tri topologique sur le graphe de dépendance des attributs. L'absence de cycle étant garanti pour les GA non circulaires, on peut appliquer cette méthode. Toutefois, elle possède des inconvénients majeurs, qui s'amplifient mutuellement. La méthode est dynamique, et coûteuse en temps.

Or, on peut déterminer statiquement une stratégie indépendante de l'arbre qui donne l'ordre de calcul des attributs [JP89]. Plus exactement, on peut construire un algorithme de parcours, éventuellement en plusieurs passes, dans lequel les attributs sont calculés de façon optimale, grâce à un ordre total [JP90] [Rou92].

On obtient par ces méthodes des séquences de visite, qui permettent de construire, pour chaque non-terminal, une fonction qui calcule ses attributs. Ces fonctions prennent en paramètre un certain état de l'arbre, et s'appellent récursivement.

Exemple

Prenons par exemple la grammaire attribuée BIRD qui remplace chaque feuille d'un arbre binaire par la plus petite de ces feuilles :

$$
\begin{array}{l}
axiom : \left\{ \begin{array}{l} Z \rightarrow T \\ Z.bird := T.bird \\ T.min_h := T.min_s \end{array} \right. \\
leaf : \left\{ \begin{array}{l} T \rightarrow A \\ T.bird := leaf(T.min_h) \\ T.min_s := A \end{array} \right. \\
node : \left\{ \begin{array}{l} T \rightarrow T_1 T_2 \\ T.bird := node(T_1.bird,\, T_2.bird) \\ T.min_s := inf(T_1.min_s,\, T_2.min_s) \\ T_1.min_h := T.min_h \\ T_2.min_h := T.min_h \end{array} \right.
\end{array}
$$

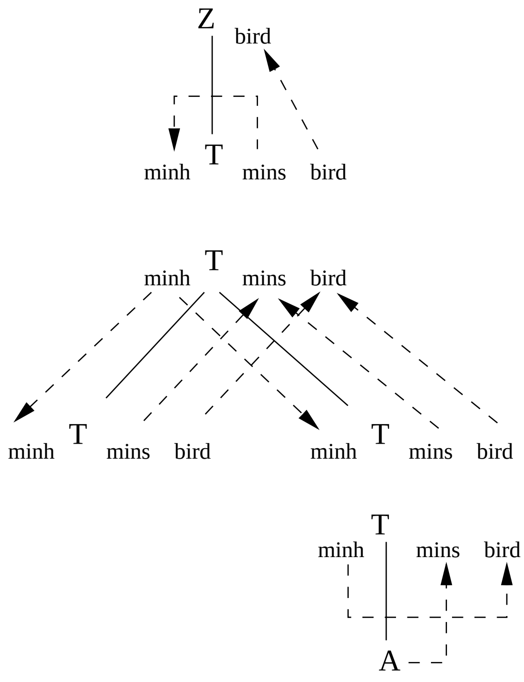

FIG. 1.2 – *Graphes de dépendance pour BIRD*

Les dépendances entre attributs (voir figure 1.2) font apparaître deux passes de calcul pour T. Dans une première passe, l'attribut min_s est calculé par une recherche en profondeur dans l'arbre. Puis, au niveau de la racine Z, sa valeur se propage dans l'arbre grâce a l'attribut min_h qui permet de synthétiser le résultat dans l'attribut $bird$.

On peut donc utiliser les séquences de visite présentée figure 1.3, qui se traduisent en CAML par le programme 1.4.

```
Z :  VISIT 1 :
       [axiom : T]
         CALL T for VISIT 1
         T.minh :=T.mins
         CALL T for VISIT 2
         Z.bird :=T.bird
     END

T :  VISIT 1 :
       [leaf : L]
         T.mins :=L
       [node : T1 T2]
         CALL T1 for VISIT 1
         CALL T2 for VISIT 1
         T.mins :=inf(T1.mins, T2.mins)
     VISIT 2 :
       [leaf : L]
         T.bird :=leaf(T.minh)
       [node :T1 T2]
         T1.minh :=T.minh
         CALL T1 for VISIT 2
         T2.minh :=T.minh
         CALL T2 for VISIT 2
         T.bird :=node(T1.bird,T2.bird)
     END
```

Fig. 1.3 – *séquence de visite pour BIRD*

```
type T = leaf of int | node of T*T ;;

let BIRD t =
    let mins = tvisit1 t in
    tvisit2 t mins
;;

let rec tvisit1 t = match t with
    leaf(L)         → L |
    node(T1,T2)     → inf (tvisit1 T1) (tvisit1 T2)
;;

let rec tvisit2 t m = match t with
    leaf(L)         → leaf(m) |
    node(T1,T2)     → node (tvisit2 T1 m, tvisit2 T2 m)
;;
```

Fig. 1.4 – *Programme BIRD en CAML*

Chapitre 2

Fnc-2

2.1 Applications en Fnc-2

Le concept de base de Fnc-2 est de définir des algorithmes qui prennent en entrée un arbre attribué, et qui donnent en sortie d'autres arbres attribués, ou avec une décoration différente. Bien entendu ces algorithmes sont décrits par des GA.

L'application est donc constituée de plusieurs passe, chaque algorithme constituant une passe. La possibilité d'enchaîner des passes donne un moyen simple d'écrire des applications modulaires.

Une application Fnc-2 se compose essentiellement de trois parties. La première est développée autour de Syntax [BD88], un générateur d'analyseurs lexico-syntaxiques. Cette partie frontale permet d'analyser un texte source passé à l'application, afin de construire un arbre abstrait. La partie centrale contient tous les évaluateurs nécessaires au calcul des attributs. Enfin, la partie finale doit être capable de produire les résultats.

La figure 2.1 indique schématiquement la structure d'une application en deux passes. Dans ce shéma la partie finale n'est pas reprensentée, car elle peut prendre des formes très différentes (effets de bord, impressions d'abres, etc). L'application est donc définie par deux grammaire attribuées, et est utilisée sur les données "data.in" pour fournir le résultat "data.out".

2.2 Spécifications en Fnc-2

Pour faire face à la diversité des objets requis pour une application complète, le système Fnc-2 dispose de plusieurs langages :

- ATC, pour décrire la syntaxe d'entrée. ATC permet de décrire une grammaire hors-contexte pour Syntax, et de construire l'arbre abstrait.
- ASX, pour décrire la structure des arbres abstraits.
- OLGA, pour décrire les grammaires attribuées. A noter qu'OLGA peut s'interfacer avec d'autres languages comme C ou CAML.
- PPAT, qui est un language d'impression pour les arbres abstraits.

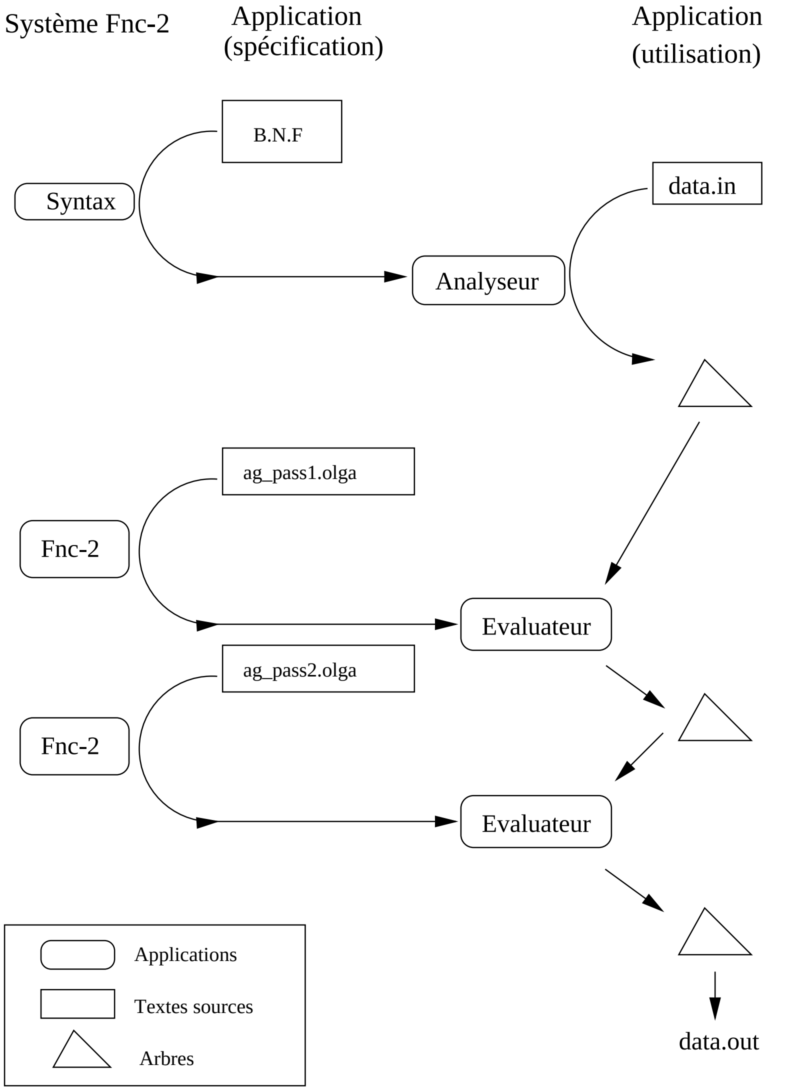

FIG. 2.1 – *une application sous Fnc-2*

Seuls ASX et OLGA seront présentés ici, car ils introduisent la notion d'opérateurs et de phyla.

Description des Arbres Abstraits (ASX)

Le formalisme de description des arbres abstraits repose sur la définition de phyla (sing. phylum) et d'opérateurs, qui sont respectivement à rapprocher des notions de non-terminaux et de productions des grammaires usuelles.

Un phylum est un type, sur lequel sont définis des opérateurs. Ces derniers sont des fonctions de construction d'arbre abstraits, dont le prototype est calqué sur une production. Prenons par exemple les deux productions :

$$G = \begin{cases} p\ : X \rightarrow AX \\ q\ : X \rightarrow B \end{cases}$$

On peut définir le phylum (type) X_t qui représente un sous-arbre dont la racine est étiquetée par X ; de même pour les phyla (types) A_t et B_t. Soit p_{op} l'opérateur qui, prenant un sous-arbre a de type A_t et un sous-arbre x de type X_t, construit un sous-arbre t de racine u tel que :

-$label(u) = X$
-$prod(u) = p$
-les fils de u soient a et x

De même, on définit q_{op}.

Alors, les arbres abstraits définis par G sont absolument identiques à ceux que l'ont peut construire avec p_{op} et q_{op}. On a là le moyen de travailler sur des structures abstraites, aussi bien que sur des arbres concrets tirés d'un texte source. C'est pourquoi, en Fnc-2, on préfère les notions de phyla et d'opérateurs aux productions.

Pour terminer avec l'exemple précédant, G se spécifie en ASX de la façon suivante :

$$\begin{cases} X_t = p_{op}, q_{op}\ ; \\ \quad p_{op} \rightarrow A_t\, X_t\ ; \\ \quad q_{op} \rightarrow B_t\ ; \end{cases}$$

L'équivalent en CAML de cette description est la définition du type suivant :

$$\text{type } X_t = q_{op} \text{ of } B_t \mid p_{op} \text{ of } A_t * X_t\ ;;$$

Ici, seul le type X_t est défini. Pour être complet, il faudrait ajouter des opérateurs pour construire des objets de type A_t et B_t. De plus, pour définir complètement la GA, il faut pour chaque opérateur donner une liste de règles sémantiques. Fnc-2 offre à ce propos des facilités au programmeur.

Dans les GA classique, chaque production a un nombre fixe de symboles en partie droite. En ASX, on peut définir des opérateurs d'arité variable. Cette facilité est très utile pour définir des listes. Du point de vue théorique et pratique, ces opérateurs n'apportent aucune complication, et on a :

$$\begin{cases} L_t = p\ ; \\ p \rightarrow A_t *\ ; \end{cases} \Leftrightarrow \begin{cases} L_t = p_{null}, p_{pre}, p_{post}\ ; \\ p_{null} \rightarrow\ ; \\ p_{pre} \rightarrow A_t\, L_t\ ; \\ p_{post} \rightarrow L_t\, A_t\ ; \end{cases}$$

Règles sémantiques(OLGA)

OLGA permet d'écrire les règles sémantiques pour chaque opérateur. OLGA contient un language fonctionnel permettant de décrire les expressions qui sont utilisées dans les règles sémantiques. C'est un language à vocation générale très applicatif :

- *types* : OLGA est fortement typé, mais le compilateur peut faire de l'inférence de type non polymorphe. En revanche, on peut définir des types polymorphes.
- *structures* : les structures d'enregistrement et d'union existent, et peuvent être mutuellement récursives. Toute structure d'arbre ou de liste est donc facilement représentable. De plus, les listes d'objets font partie des types de base. Ainsi, quand on définit le type $A = list\, of\, B$, le compilateur génère automatiquement les constructeurs adéquats, ainsi qu'un certain nombre d'outils (test de liste vide, longueur de liste, tête et queue de liste...).
- *applicatif* : OLGA ne permet pas d'effet de bord, et en fait, il n'y a pas de séquence d'instructions. C'est un langage fonctionnel pur dans lequel les boucles sont remplacées par la récursion. Mais le compilateur est capable d'éliminer les récursions terminales dans un souci d'optimisation. Les fonctions peuvent également être mutuellement récursives.
- *variables* : les seules variables qui existent en OLGA sont les paramètres des fonctions, ainsi que les variables locales définies par l'opérateur $let\, x = e_1\, in\, e_2$. Il n'y a pas de variables globales, ce qui va de paire avec l'absence d'effet de bord.
- *pattern matching* : en complément de la structure conditionnelle classique, l'opérateur *case* permet de faire du pattern matching comme en CAML.
- *interfaces* : OLGA peut faire appel à des fonctions externes écrites séparément, en C par exemple. On peut par cet artifice produire certains effets de bord, comme afficher un message d'erreur.

Une présentation complète du language OLGA est disponible dans [JLP90].

Dans la pratique, de nombreuses règles sémantiques sont de simples copies du type $X.a := Y.a$, qui ne font que propager dans l'arbre attribué un résultat. Par exemple, une table de symboles doit être propagée dans tous les nœuds d'une expression arithmétique. Fnc-2 génère automatiquement toutes ces règles de copie “implicites”.

Un autre cas très courant est l'utilisation d'un attribut qui est en fait la “réunion” de deux attributs, l'un synthétisé, et l'autre hérité. Par exemple, dans une suite de déclarations, il faut deux tables de symboles. L'une est de type hérité, et contient les déclarations précédentes. L'autre, de type synthétisé, tient compte de la déclaration courante. Fnc-2 possède un type d'attribut “compound” qui permet de déclarer en une seule fois ces deux attributs. Les règles de copie implicites sont alors générées automatiquement.

Chapitre 3

Généricité & Couplage

Ce chapitre reprend une bonne partie de [BJPR93], avec quelque modifications. Ainsi, j'ais levé certaines restrictions sur la relation de correspondance, et réécris les algorithmes 3.3 et 3.2. Enfin, la partie 3.7.2 et les algorithmes 3.5 et 3.6 n'étaient pas présents dans l'article initial.

3.1 Grammaires Attribuées Fonctionnelles & Méta-Composition

On peut voir certaines grammaires attribuées comme des fonctions, et ainsi définir une composition symbolique.

Supposons que nous ayons deux grammaires, G_1 et G_2 de racines Z_1 et Z_2, ainsi qu'une grammaire attribuée $\alpha(G_1, A_1, F_1)$. Supposons de plus que $A(Z_1)$ soit réduit à un seul attribut, synthétisé, qui soit du type Z_2 (un non-terminal étant en fait un type d'arbre). Dans cette situation, le calcul de la valeur de cet attribut revient à définir un morphisme des arbres abstraits de G_1 sur ceux de G_2 :

$$G_1 \xrightarrow{\alpha} G_2$$

On appelle α une grammaire attribuée fonctionnelle.

Composition

Bien entendu, si on a une autre grammaire attribuée fonctionnelle β :

$$G_2 \xrightarrow{\beta} G_3$$

on peut définir leur composition $\gamma = \beta \circ \alpha$ par :

$$G_1 \xrightarrow{\alpha} G_2 \xrightarrow{\beta} G_3$$

Cette opération formelle tout à fait classique n'est pas très intéressante en pratique. Si elle donne l'illusion de fusionner deux GAs en une seule, pour faire les calculs, on est bien obligé de construire l'arbre intermédiaire sur G_2. C'est une opération coûteuse, surtout si des simplifications sont envisageables.

Notamment, si α vérifie $\alpha \circ \alpha = Id$ on fait deux calculs qui ne servent à rien. Il convient donc de réellement fusionner les deux GAs α et β.

En fait, on peut construire une grammaire attribuée qui soit équivalente à la composition symbolique, mais qui ne construit pas l'arbre intermédiaire sur G_2. Cette opération est appelée méta-composition.

Méta-Composition : un aperçu

La méta-composition est maintenant complètement intégrée au système Fnc-2, et une description complète de cet opérateur est donnée dans [Rou94]. Les principes utilisés sont du type suivant. Imaginons que dans α on ait :

$$\begin{array}{l} p \,:\, X_0 \to \ldots X_i \ldots \\ \quad a(X_0) \,:=\, f(\ldots, a_j(X_i) \ldots) \\ \ldots \end{array}$$

et que dans β on ait :

$$f \,:\, Y_0 \to \ldots Y_j \ldots$$

alors :

- si s est un attribut synthéthisé de Y_0, on a dans les règles sémantiques de f une définition du type $s(Y_0) \,:=\, exp$ (une expression). Alors on ajoute l'attribut $a.s$ à X_0, et la règle sémantique $a.s(X_0) \,:=\, exp$ à celles de l'opérateur p.
- si h est un attribut hérité de Y_j, on a dans les règles sémantiques de f une définition du type $h(Y_j) \,:=\, exp$. Alors on ajoute à X_i l'attribut $a_j.h$, et la règle sémantique $a_j.h(X_i) \,:=\, exp$ à celles de p.

Bien entendu, les expressions exp sont modifiée en conséquence.

Les autres cas sont un peu plus compliqués, mais sont gérés sur le même principe. Prenons un exemple pour se fixer les idées :

Si on a :

$$\begin{array}{l} p \,:\, X \to A\,B \\ \quad a(X) \,:=\, f(A, b(B)) \end{array}$$

et :

$$\begin{array}{l} f \,:\, E_0 \to E_1\,E_2 \\ \quad s(E_0) \,:=\, s(E_1) + s(E_2) \\ \quad h(E_1) \,:=\, h(E_0) \\ \quad h(E_2) \,:=\, h(E_0) \end{array}$$

la méta-composition donne :

$$\begin{array}{l} p \,:\, X \to A\,B \\ \quad a.s(X) \,:=\, s(A) + b.s(B) \\ \quad h(A) \,:=\, h(X) \\ \quad b.h(B) \,:=\, h(X) \end{array}$$

La notion de méta-composition se rapproche de celle de déforestation en programmation fonctionnelle.

3.2 Grammaires Attribuées Génériques

Si une GA spécifie un algorithme particulièrement optimisé et testé, il devient très intéressant de pouvoir le réutiliser dans d'autres programmes. Plus généralement, si on définit un ensemble d'attributs avec leurs règles sémantiques, on a envie de pouvoir transférer cette sémantique sur une autre grammaire. On parle alors de GA générique, ou de gène par abus de language.

Deux problèmes pratiques se posent pour rendre cette notion utilisable. Le premier est de trouver un moyen de spécifier comment le gène doit être instancié sur une grammaire particulière. Le second, mais qui lui est très lié, est de savoir comment écrire un gène pour que cette instanciation se passe sans problème.

Dans ce chapitre, il sera question de l'instanciation du gène. Les propriétés de ce dernier ne seront étudiées qu'au chapitre suivant.

3.3 Stratégie d'Instanciation

La méthode générale a déjà été évoquée en introduction, mais nous la reprenons plus en détails ici. Notons que l'on trouve des idées similaires dans [GG84] et [FMY92].

Le gène, AG_g, est capable de faire un calcul, sous forme de calcul d'attributs, sur un arbre abstrait d'une grammaire G_o. Imaginons, pour se fixer les idées, que AG_g soit fonctionnelle. Le programmeur qui désire utiliser l'algorithme AG_g ne travaille pas sur la grammaire G_o mais sur une grammaire G_i. Il est évident que, pour qu'une instanciation soit possible, il faut que les structures du gène se retrouvent dans G_i. Mais pour disposer d'un grand pouvoir d'expression, il faut permettre à ces structures d'être éparpillées dans G_i, avec des noms différents, et parmi bien d'autres structures sur lesquelles AG_g ne doit pas s'appliquer.

L'idée de base est de partir d'une relation, spécifiée par l'utilisateur, qui donne la correspondance que l'on souhaite entre les types de G_o et certains types de G_i. A partir de cette relation, il sera possible de générer une grammaire attribuée fonctionnelle, AG_{coup}, dite "de couplage", dont le rôle est le suivant. Si on dispose d'un arbre abstrait a sur G_i, l'application de AG_{coup} sur a donne un arbre abstrait a' sur G_o qui représente la structure éparpillée dans a sur laquelle on veut appliquer AG_g.

L'étape suivante consisterait à appliquer AG_g sur a', et l'on obtiendrait exactement ce que l'on veut. Mais pour éviter des calculs inutiles, on va plutôt utiliser l'opérateur de méta-composition, pour calculer :

$$AG_{inst} = AG_g \otimes AG_{coup}$$

Ainsi, on ne construit pas l'arbre abstrait intemédiaire a' et les attributs de AG_g sont directement calqués sur G_i et leur calcul est propagé dans a.

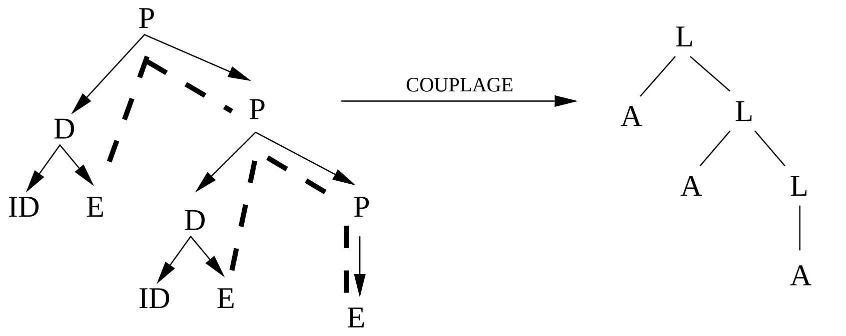

FIG. 3.1 – *Exemple de couplage*

Exemple

Prenons un exemple simple. On dispose d'un algorithme générique pour calculer l'inversion d'une liste, et on souhaite s'en servir pour inverser la liste des expressions d'un programme. Le gène est décrit par la grammaire attribuée suivante :

$$INV : \left\{ \begin{array}{l} axiome\ : Z \rightarrow L \\ \quad L.h\ := nil() \\ \quad Z.INV\ := L.s \\ cons\ : L \rightarrow A\, L' \\ \quad L'.h\ := cons(A, L.h) \\ \quad L.s\ := L'.s \\ one\ : L \rightarrow A \\ \quad L.s\ := cons(A, L.h) \\ nil\ : L \rightarrow . \\ \quad L.s\ := L.h \end{array} \right.$$

Le programme dont on veut inverser la liste des expressions est décrit par la grammaire suivante :

$$PROG : \left\{ \begin{array}{l} start\ : S \rightarrow P \\ decl\ : P \rightarrow D\, P \\ prog\ : P \rightarrow E \\ def\ : D \rightarrow ID\, E \\ (\ldots)E \rightarrow (\ldots) \end{array} \right.$$

On voit que les expressions se trouvent dans les déclarations sous le non-terminal D, et en fin de programme (voir figure 3.1). On va donc dans un premier temps faire remonter les expressions des déclarations jusqu'au non-terminal P, et ensuite construire la liste adéquate. Une grammaire attribuée de couplage possible est :

$$COUP:\left\{\begin{array}{l} start\,:S\rightarrow P\\ \quad S.Z\,:=axiome(P.L)\\ decl\,:P\rightarrow D\,P'\\ \quad P.L\,:=cons(D.exp,P'.L)\\ prog\,:P\rightarrow E\\ \quad P.L\,:=one(E)\\ def\,:D\rightarrow ID\,E\\ \quad D.exp\,:=E \end{array}\right.$$

L'attribut *exp* de D sert à la remontée des expressions, et l'attribut L de P sert à construire la liste des expressions.

La méta-composition fait alors une transposition des règles sémantiques de INV aux endroits de $COUP$ où on construit des morceaux d'arbres de INV. Ainsi, quand pour l'opérateur *decl* on construit le sous-arbre $cons(D.exp, P'.L)$, on applique les règles sémantiques de *cons*. Cela donne :

$$INV\otimes COUP:\left\{\begin{array}{l} start\,:S\rightarrow P\\ \quad P.L.h\,:=nil()\\ \quad S.Z.INV\,:=P.L.s\\ decl\,:P\rightarrow D\,P'\\ \quad P'.L.h\,:=cons(D.exp,P.L.h)\\ \quad P.L.s\,:=P'.L.s\\ prog\,:P\rightarrow E\\ \quad P.L.s\,:=cons(E,P.L.h)\\ def\,:D\rightarrow ID\,E\\ \quad D.exp\,:=E \end{array}\right.$$

On voit que la construction de la liste n'est plus nécessaire, et que l'algorithme a été adapté à la structure un peu particulière du programme, qui cache une partie de ses expressions dans des déclarations. On a donc effectué une véritable instanciation de INV sur $PROG$, et la grammaire attribuée obtenue est similaire à celle que l'on aurait écrite à la main en se creusant la tête.

Nous allons voir maintenant comment obtenir $COUP$, la grammaire attribuée de couplage.

3.4 Relation de Correspondance

Une relation de correspondance Cor entre $G_i = (N_i, T_i, Z_i, P_i)$ et $G_o = (N_o, T_o, Z_o, P_o)$ est une application de $N_o \cup T_o$ dans $\mathcal{P}(N_i \cup T_i)$ telle que :

$$\forall x, y \in N_o \cup T_o, Cor(x) \cap Cor(y) = \emptyset$$

$$Cor(Z_o) = \{Z_i\}$$

On définit aussi, sur la grammaire G_i la relation $\Rightarrow$ sur $N_i \cup T_i$ telle que :

$$(X \Rightarrow Y)\, ssi\, (p : X \rightarrow X_0 \ldots Y \ldots X_n \in P_i)$$

On note enfin $\overset{*}{\Rightarrow}$ la fermeture transitive de $\Rightarrow$. Avec ces relations, on peut définir les ensembles suivants, qui nous serons très utiles par la suite :

- $S_i = N_i \cup T_i$ et $S_o = N_o \cup T_o$, les symboles de G_i et G_o.
- $S_k = \{X \in S_i \mid \exists Y \in S_o, X \in Cor(Y)\}$ est l'ensemble des symboles clefs. Ce sont les éléments qui forment la structure sur laquelle on veut appliquer AG_g.
- $N_n = \{X \in N_i - S_k \mid \exists Y \in S_k, X \overset{*}{\Rightarrow} Y\}$ est l'ensembles des symboles neutres. C'est par l'intermédiaire de ces non-terminaux que les symboles clefs sont regroupés pour pouvoir former un arbre intermédiaire sur G_o.
- $S_d = S_i - (S_k \cup N_n)$ est l'ensembles des symboles morts, qui ne jouent aucun rôle pour l'application de AG_g.

On peut s'inquiéter à ce stade du peu de distinction entre terminaux et non-terminaux. En fait, une définition antérieure de la relation de correspondance imposait $Cor(X) \subset N_i \, ssi \, X \in N_o$ et $Cor(X) \subset T_i \, ssi \, X \in T_o$. On imposait même que chaque type de G_o soit mis en correspondance avec au moins un type de G_i. Mais cela réduisait le pouvoir d'expression de la méthode de couplage. Ces restrictions ont été levées et nous en reparlerons à propos de l'écriture d'une GA générique.

3.5 Réduction en Arbres Clos

Après avoir défini une correspondance entre les symboles de G_i et G_o, on souhaite trouver une correspondance entre des morceaux d'arbres de G_i et des productions de G_o. C'est l'idée qui consiste à récolter des parties de la structure qui nous intéresse au travers d'un arbre sur G_i.

Définition

Un arbre clos est un sous-arbre de G_i tel que sa racine et toutes ses feuilles soient étiquetés par des symboles de S_k, et tel que tous les nœuds internes soient étiquetées par des symboles de N_n.

Pour calculer ces arbres clos, un certain nombre de transformations et de vérifications sont nécessaires. La première consiste bien sûr à "nettoyer" G_i de tous les symboles morts (de S_d). Mais le plus important est d'assurer que le calcul des arbres clos se termine. En effet, la plupart des grammaires sont récursives, c'est à dire que des non-terminaux figurent à la fois en partie droite et en partie gauche d'une même production. Ces productions ont la particularité d'engendrer des sous-arbres de taille arbitraire, donc en nombre infini.

Elimination des récursions

Fort heureusement, les récursions entre non-terminaux de S_k ne posent pas de problème, car un arbre clos ne peut pas avoir de nœud interne étiqueté par un symbole dans S_k.

En revanche, les récursions sur des symboles neutres sont problématiques. Si on a par exemple les productions suivantes :

$$\begin{cases} X \rightarrow Y \\ Y \rightarrow Y\,Y \\ Y \rightarrow X \end{cases}$$

avec $X \in S_k$ et $Y \in N_n$, alors les arbres clos que l'on peut construire sont en nombre infini. Cela signifie que le non-terminal X se dérive en une infinité de X sous forme d'arbre binaire. Si par ailleurs, X est un symbole clef, cette structure doit exister dans G_o, et donc il n'est pas normal que Y soit neutre. Dans ce type de cas, on génère une erreur, car de toute façon le calcul des arbres clos ne terminerait pas.

De manière plus générale, on élimine les récursions sur les symboles neutres par l'algorithme suivant :

1. On définit tout d'abord la relation $\overset{*}{\underset{Cor}{\Rightarrow}}$ par :

$$X \overset{*}{\underset{Cor}{\Rightarrow}} Y \Leftrightarrow \begin{cases} X \Rightarrow A_1 \Rightarrow \dots A_n \Rightarrow Y \\ \forall i, A_i \in N_n \\ X \in N_n \end{cases}$$

2. On définit alors l'ensemble N_r des non-terminaux neutres et récursifs par :

$$N_r = \{X \in N_n \mid X \overset{*}{\underset{Cor}{\Rightarrow}} X\}$$

3. Sur N_r on définit la relation d'équivalence Γ telle que :

$$X\,\Gamma\,Y \Leftrightarrow (X \overset{*}{\underset{Cor}{\Rightarrow}} Y\; et\, Y \overset{*}{\underset{Cor}{\Rightarrow}} X)$$

4. Tous les symboles de N_r sont alors renommés par leur classe d'équivalence, et les productions de P_i sont modifiées en conséquence.

5. Toutes les productions $p : X \rightarrow X$ avec $X \in N_n$ sont éliminées. En effet, ces productions ne peuvent pas modifier les frontières d'un arbre clos.

6. S'il reste au moins une production où un terminal neutre figure en partie droite et en partie gauche, générer une erreur, le couplage est impossible.

Si cet algorithme ne génère aucune erreur, alors nous aurons la garantie que le calcul des arbres clos se termine.

Calcul des Arbres Clos

L'idée de cet algorithme est de partir de sous-arbres dont la racine est étiquetée par un symbole clef, et de remplacer les feuilles neutres par des sous-arbres, jusqu'à obtenir tous les arbres clos. Pour alléger la description de l'algorithme, on note $t \overset{u,p}{\rightarrow} t'$ l'opération qui consiste à transformer l'arbre t en remplaçant la feuille u par un sous-arbre compatible avec la production p.

$A \leftarrow \{t \mid \exists p \in P_i, LHS(p) \in S_k, t = p(u_1, \ldots, u_n)\}$
$C \leftarrow \emptyset$
ep **r**
  soit $t \in A$
  si t est clos **alors**
    $C \leftarrow C \cup \{t\}$
  sinon
    soit $u \in leaves(t)$ **tel que** $label(u) \in N_n$
      pour tout $p \in P_i$ **tel que** $RHS(p) = label(u)$
        $t \overset{u,p}{\rightarrow} t'$
        $A \leftarrow A \cup \{t'\}$
  fin si
  $A \leftarrow A - \{t\}$
tant que $A \neq \emptyset$
$C = \{$ arbres clos $\}$

FIG. 3.2 – *Calcul des Arbres Clos*

L'algorithme est décrit figure 3.2. Il se termine dès que l'ensemble d'arbres A est vide. Pendant une itération, soit A perd un élément, soit un arbre t subit une ou plusieurs transformations $t \overset{u,p}{\rightarrow} t'$, et les arbres ainsi obtenus remplacent t dans A. Or, grâce aux transformations effectuées plus haut, un arbre t ne peut subir qu'un nombre fini de transformations $t \overset{u,p}{\rightarrow} t'$, et comme A est fini au début de l'algorithme, A devient vide en un temps fini.

3.6 Instanciation

Maintenant que nous disposons d'un ensemble d'arbres clos, il reste à les associer à leurs équivalents dans la grammaire du gène. Plusieurs solutions sont possibles, mais la plus simple est la suivante. Soit t un arbre clos, x sa racine, et $u_1 \ldots u_n$ ses feuilles de gauche à droite ; on appelle production virtuelle associée à t la production $x \rightarrow u_1 \ldots u_n$. Si cette production fait partie de la grammaire du gène, on associe l'opérateur correspondant à l'arbre. Sinon, on peut dans un premier temps considérer que l'instanciation n'est pas possible. En fait, des extensions sont possibles, mais nous en reparlerons au moment de l'écriture du gène.

Nous avons maintenant un ensemble de paires (t, op) associant un arbre clos et un opérateur (c-a-d une production) de la grammaire du gène. Rappelons que l'on cherche à construire un arbre abstrait de G_o stocké dans un attribut de la racine de G_i. La méthode consiste alors, pour une paire (t, op), à faire remonter les symboles clefs qui sont des feuilles de t à la racine de t, puis à synthétiser avec op le sous-arbre associé. Cette opération est assurée par la grammaire attribuée de couplage, que l'on peut générer automatiquement à partir des paires (t, op).

3.7 Génération de la GA de Couplage

Pour en clarifier la présentation, la génération de la GA de couplage est scindée en deux parties. Dans la première partie, nous verrons comment on peut sur chaque paire (t, f) écrire une partie de cette GA. Il y aura souvent plusieurs solutions possibles pour chaque paire. Pour que la construction soit possible, il faut donc choisir une solution partielle pour chaque paire, telle que toutes ces parties de GA soient compatibles entre elles. Dans la pratique, ces deux opérations se font simultanément grâce à une petite modification des algorithmes que nous allons décrire.

3.7.1 GA de Couplage pour une paire (t, f)

Notations

On adoptera les notations suivantes :

- Z est le non-terminal clef qui étiquette la racine de t.
- $A_1 \dots A_n$ sont les feuilles de t.
- f est un opérateur.
- si Y est un symbole de G_i, y est le symbole de G_o tel que $Y \in Cor(y)$.
- si u est un nœud de t, X et p désignent le symbole et la production qui étiquettent u. op est l'opérateur associé à p.
- chaque symbole clef ou neutre de G_i va recevoir une liste d'attributs. Ces attributs ont pour type un phylum de G_o. Pour simplifier, un attribut de type y associé à un symbole X de G_i aura pour nom $y_j(X)$, où j est un indice permettant de distinguer plusieurs attributs ayant le même type. Parfois, on notera simplement y_j si aucune confusion n'est possible.
- $A_t(X)$ est la liste des attributs de X dans l'arbre clos t.
- $F_t(op)$ est la liste des règles sémantiques pour l'opérateur op.

Attributs des symboles clefs

La GA de couplage doit être fonctionnelle et doit produire un unique arbre sur G_o. On doit donc avoir :

$$\forall X \in S_k, A_t(X) = \{x_1\}$$

Fonctions $AP(u)$ et $FP(u)$

La fonction AP, définie pour chaque nœud u de t donne la liste des attributs (renumérotés) qui existent chez les fils de u. FP donne simplement la liste des règles de copie adéquates. Le calcul exact de AP et FP est donné figure 3.3.

Dans la suite, les éléments de $AP(u)$ sont notés $a^1 \dots a^n$ dans l'ordre. L'exposant ne doit pas être confondu avec les indices d'attributs. Ainsi on a pour tout k, $a^k = x_i$ pour certains i et x.

calcul de $AP(u)$ **and** $FP(u)$

$u\ :[p,op]\,X\rightarrow X_1\ldots X_n$
$AP\leftarrow\emptyset$
$FP\leftarrow\emptyset$
pour chaque $i\in[1..n]$
pour chaque $x_k\in A_t(X_i)$
soit $k'=Card\{j\mid x_j\in AP\}+1$
$AP\leftarrow append(AP,x_{k'})$
$FP\leftarrow FP\cup\{x_{k'}:=x_k(X_i)\}$
$AP(u)=AP$
$FP(u)=FP$

FIG. 3.3 – *Calcul de AP et FP*

Construction élémentaire en u

$$u\ :[p,op]\,X\rightarrow X_1\ldots X_n$$

1. Si l'arbre a déjà été construit en un nœud inférieur dans t, on a $AP(u)=\{z_1\}$ et donc il suffit de propager cet attribut :

$$A_t(X)=\{z_1\},Ft(op)=\{z_1:=z_1(X_1)\}$$

2. Si $length(AP(u))=arity(f)$, alors on a récolté tous les attributs nécessaires pour construire l'arbre. On a néanmoins le choix entre deux solutions. La première consiste à ne pas construire l'arbre tout de suite, et à transmettre les attributs au nœud supérieur :

$$(a)\quad A_t(X)=AP(u),Ft(op)=FP(u)$$

Cette solution n'est évidement pas acceptable si u est la racine de t. La seconde solution consiste à construire l'arbre tout de suite :

$$(b)\quad A_t(X)=\{z_1\},Ft(op)=\{z_1:=f(a^1,\ldots,a^n)\}$$

3. Si $length(AP(u))<arity(f)$, alors on ne peut pas construire l'arbre, et il faut transmettre les attributs au nœud supérieur :

$$A_t(X)=AP(u),Ft(op)=FP(u)$$

Notons que, d'après la construction des paires (t,f), ce cas ne peut jamais se produire à la racine.

On obtient ainsi un ensemble de solutions partielles pour chaque paire (t,f). Il faudrait normalement explorer les différentes combinaisons possibles jusqu'à ce que l'on trouve un assemblage de solutions partielles cohérentes entre elles. Mais en fait, on peut mener les deux opérations de front par un mécanisme de *backtracking*.

3.7.2 Construction & Choix des solutions Simultanés

Les notations sont les mêmes que précédement, et les fonctions AP et FP sont toujours d'usage.

Pour distinguer deux solutions, on peut remarquer qu'il suffit de connaître la profondeur de t à laquelle on synthétise z_1.

Supposons que l'on ait une solution partielle, c'est-à-dire des ensembles $A = \bigcup A(X)$ d'attributs et $F = \bigcup F(op)$ de règles sémantiques, qui soient compatibles avec un ensemble de paires (t^i, f^i). Il suffit de modifier l'algorithme de construction élémentaire pour qu'il choisisse la solution compatible avec A et F.

Pour cela, quand on cherche une solution pour la paire (t, f), la recherche est paramétrée par une borne supérieure de la profondeur à laquelle on peut construire l'arbre.

Pendant la génération de $Ft(op)$ et $At(X)$ pour un nœud de u, trois cas peuvent se présenter :

- Soit aucune règle n'existe encore pour op dans F. Dans ce cas, on utilise l'algorithme normal, à cette modification près : si on est à une profondeur trop grande, on ne cherche pas à construire l'arbre, et on ne fait que des copies.
- Soit $F(op)$ correspond à la construction de l'arbre. Il faut alors vérifier que la construction est compatible avec t, et que la profondeur n'est pas trop grande.
- Soit $F(op)$ correspond à une copie d'attributs, et on vérifie que ces copies sont compatibles avec t.

Si aucune solution n'est trouvée, c'est que l'on a fait un mauvais choix dans la solution partielle précédente. Il faut alors refaire l'exploration précédente avec une profondeur maximale diminuée. Si cette profondeur devient nulle, il faut encore revenir à la solution partielle antérieure.

3.7.3 Description détaillée

La fonction $EXPLORE$ prend en entrée les paramètres $(A, F, L, pmax)$ et renvoie (A', F') ou $erreur$ avec :

-L est une liste de paires (t^i, f^i), non encore explorées.

-A et F est une solution partielle ne tenant pas compte de L.

-A' et F' est une solution partielle compatible avec la première paire de L et la solution partielle précédante.

-$pmax$ est une limite supérieure à la profondeur autorisée pour synthétiser l'arbre.

$EXPLORE$ utilise en fait la fonction $SCAN$ qui explore effectivement les solutions possibles pour un sous-arbre de t. $SCAN$ prend les paramètres $(A, P, u, p, pmax)$ où u est un nœud de t, p sa profondeur, les autres paramètres étant ceux de $EXPLORE$. $SCAN$ retourne :

-$(erreur, A, P, p)$ si la construction est impossible à la profondeur p.

fonction $EXPLORE(A, F, L, pmax)$
  si $empty(L)$ **retourner** (A,F,pmax)
  si $pmax = 0$ **retourner** $erreur$
  $(t, f) \leftarrow head(L)$
  $(s, A', P', pmax') \leftarrow SCAN(A, F, t, 0, pmax)$
  si $s \neq erreur$ **alors**
    $(s, A'', P'', pmax'') \leftarrow EXPLORE(A', F', tail(L), +\infty)$
    si $s \neq erreur$ **alors**
      return $(s, A'', P'', pmax'')$
    sinon
      $EXPLORE(A, F, L, pmax'')$
    fin si
  sinon
    $EXPLORE(A, F, L, pmax')$
  fin si

FIG. 3.4 – *Fonction EXPLORE*

-$(cons, A', P', pmax')$ si la construction est faite à la profondeur $pmax'$.
-$(copy, A', P', pmax)$ si à la profondeur p, il n'y a que des règles de copie.

L'algorithme de $SCAN$ est décrit figure 3.5. Son fonctionnement est très simple. Quand on est dans un nœud u d'un arbre clos, soit $F(op)$ n'est pas encore défini, soit $F(op)$ contient des règles, et de ce fait, $A(X)$ contient des attributs. Dans le premier cas, on utilise l'algorithme standard de construction. Dans le second, il faut vérifier que l'on peut utiliser les mêmes règles et attributs en u.

$SCAN$ fait donc appel à trois autres fonctions, $SCAN_{NEXT}$, $SCAN_{COPY}$ et $SCAN_{CONS}$, dont les algorithmes sont décrits figure 3.6. Elles sont chargées respectivement de lancer $SCAN$ sur tous les fils d'un nœud, de vérifier qu'une copie, ou qu'une construction, sont compatibles avec les solutions précédentes.

On explore les solutions partielles en utilisant la fonction $EXPLORE$, décrite figure 3.4. Initialement, si L_0 est la liste complète des paires (t, f), on appelle :

$$EXPLORE(\emptyset, \emptyset, L_0, +\infty)$$

fonction $SCAN(A, F, u, p, pmax)$

  si u est une feuille **alors**

    retourner $(copy, A, F, min(p, pmax))$

  selon le cas :

    si $F(u) = \{$ copies $\}$:

      retourner $SCAN_{COPY}(A, F, v, p+1, pmax)$

    si $F(u) = \{$ construction d'arbre $\}$:

      si $p < pmax$

        alors retourner $SCAN_{CONS}(A, F, v, p+1, pmax)$

        sinon retourner $(erreur, A, F, pmax)$

    si $F(u) = \emptyset$:

      $(s, A, F, pmax) \leftarrow SCAN_{NEXT}(A, F, u, p, pmax)$

      si $s = erreur$ **retourner** (s,A,F,p)

      $AP(u) = \{a^1, \ldots, a^n\}$

      si $n = arity(f)$ **et** $p < pmax$ **et** $s = copy$ **alors**

        $A(X) \leftarrow \{z_1\}$

        $F(op) \leftarrow \{z_1 \leftarrow f(a^1, \ldots, a^n)\}$

        retourner $(cons, A, F, p)$

      sinon

        $A(X) \leftarrow AP(u)$

        $F(op) \leftarrow FP(u)$

        retourner $(copy, A, F, pmax)$

      end si

  end case

FIG. 3.5 – *Fonction SCAN*

fonction $SCAN_{NEXT}(A, F, u, p, pmax)$
  $l \leftarrow fils(u)$
ep **r**
    $(s, A, F, pmax) \leftarrow SCAN(A, F, head(l)), p+1, pmax)$
    $l \leftarrow tail(l)$
  tant que $s \neq erreur$ **et** l non-vide
  retourner $(s, A, F, pmax)$

fonction $SCAN_{COPY}(A, F, u, p, pmax)$
  si u est une feuille **alors**
    retourner $(copy, A, F, pmax)$
  $(s, A', F', p') \leftarrow SCAN_{NEXT}(A, F, u, p, pmax)$
  si $s = erreur$ **ou** $s = cons$
    ou $A(X) \neq AP(u)$ **ou** $F(op) \neq FP(u)$
  alors
    retourner $(erreur.A, F, p)$
  sinon
    retourner (s, A', F', p')

fonction $SCAN_{CONS}(A, F, u, p, pmax)$
  si u est une feuille **alors**
    retourner $(erreur, A, F, pmax)$
  $(s, A', F', p') \leftarrow SCAN_{NEXT}(A, F, u, p, pmax)$
  si $s = erreur$ **ou** $A(X) \neq \{z_1\}$
    ou $F(op) \neq \{z_1 \leftarrow f(\ldots)\}$
  alors
    retourner $(erreur.A, F, p)$
  sinon
    retourner (s, A', F', p')

FIG. 3.6 – *Fonctions $SCAN_{NEXT}$, $SCAN_{COPY}$ et $SCAN_{CONS}$*

Chapitre 4

Écriture d'un Gène

Dans le chapitre précédent, nous avons vu comment créer automatiquement une grammaire attribuée de couplage, première phase de la réutilisation d'un algorithme générique. Il nous reste quelques problèmes en suspens, comme les définitions sémantiques pour les terminaux du gène, et l'association des productions virtuelles aux opérateurs du gène.

Dans ce chapitre, je donne une réponse au problème, soulevé dans [BJPR93], sur la manière d'écrire un gène pour qu'il soit instanciable sur le plus grand nombre de grammaires possible. Je définis donc une forme standard (vis à vis du couplage) pour les grammaires attribuées, et une extension de la relation de correspondance.

4.1 Problèmes de Sémantique...

Un arbre abstrait représente une organisation d'une suite de terminaux. On a donc deux types d'informations dans l'arbre, dans les feuilles, et dans les nœuds internes. Dans un arbre attribué, il existe une information supplémentaire dans les attributs de chaque nœud. Attributs et terminaux sont deux façons équivalentes de stocker de l'information dans l'arbre. Prenons un exemple. Voici deux grammaires attribuées équivalentes, candidates pour devenir des algorithmes génériques de calcul d'expressions arithmétiques :

$$G_1 : \left\{ \begin{array}{l} E = add, mul, nbr\,; \\ synthesized\ val(E)\ : int\,; \\ add \rightarrow E_1 E_2\,; \\ \quad val\ := val(E_1) + val(E_2)\,; \\ mul \rightarrow E_1 E_2\,; \\ \quad val\ := val(E_1) * val(E_2)\,; \\ nbr \rightarrow\,; \\ \quad val\ := \ldots\,; \end{array} \right. \qquad G_2 : \left\{ \begin{array}{l} E = add, mul, nbr\,; \\ synthesized\ val(E)\ : int\,; \\ add \rightarrow P E_1 E_2\,; \\ \quad val\ := val(E_1) + val(E_2)\,; \\ mul \rightarrow M E_1 E_2\,; \\ \quad val\ := val(E_1) * val(E_2)\,; \\ nbr \rightarrow int\,; \\ \quad val\ := int\,; \\ M = `*`\,; \\ P = `+`\,; \end{array} \right.$$

Dans G_1, la valeur de l'attribut $val(nbr)$ est issue d'un calcul précédent. Dans G_2, le type int est un phylum terminal. La différence est de taille, car dans le premier exemple, l'opérateur nbr assimile le type E à sa valeur numérique. Dans le deuxième exemple, l'opérateur nbr exprime le fait qu'une expression arithmétique peut être un nombre, c'est-à-dire un terminal de type int.

De plus, pour G_1, l'analyseur syntaxique ne produirait pas un arbre abstrait, mais un arbre partiellement attribué. Cela signifie que les opérateurs de G_1 ne permettent pas, seuls, de construire un arbre pour G_1. Il manquerait les valeurs des attributs val aux feuilles. En revanche, ce problème ne se pose pas pour G_2, où ces attributs sont remplacés par des terminaux.

Avant d'aller plus loin, comparons un peu plus ces deux grammaires. Dans G_1, l'opérateur add et l'opérateur mul ont tous les deux la même production associée. En effet ce sont deux opérateurs de prototype $E \times E \to E$. De ce fait, l'algorithme que nous avons donné va échouer quand il faudra associer un arbre clos de production virtuelle $E \to E\,E$ à un opérateur du gène. Cette ambiguïté ne peut pas être levée autrement que par une spécification du programmeur.

Voila donc deux raisons qui font que G_1, bien que plus élégante que G_2, ne peut pas être un gène. On voit ainsi apparaître un réel problème. Dans les grammaires attribuée une information sémantique a plusieurs représentations qui ne sont pas toutes assimilables par l'opérateur de couplage.

4.2 Correspondance Etendue

Quand bien même G_2 serait choisit pour le gène, on aimerait bien instancier les algorithmes définis pour G_2 sur G_1. La solution est simple, car il suffit d'étendre la définition de la relation de correspondance.

Si une information se présente sous forme d'un terminal dans le gène, on peut avoir à la mettre en correspondance avec un attribut. Ainsi, on a envie de pouvoir spécifier, pour l'exemple précédent, que $Cor(int) = \{val(nbr)\}$.

Certains terminaux d'un gène n'ont aucun rôle, sinon de pouvoir distinguer deux productions. C'est le cas de P et M dans G_2. On peut donc vouloir spécifier que ces terminaux (sans valeur sémantique) sont en correspondance avec des opérateurs. Ainsi, on a envie d'écrire pour l'exemple précédant que $Cor(P) = \{add\}$. On note $\mathcal{A}$ l'ensemble de ces terminaux.

Une relation de correspondance étendue Cor_E entre $G_i = (PH_i, OP_i, A_i, F_i)$ et $G_o = (PH_o, OP_o, A_o, F_o)$ est une application de PH_o dans $\mathcal{P}(PH_i \cup OP_i \cup A_i)$ telle que :

$$\forall x, y \in PH_o, \quad Cor_E(x) \cap Cor_E(y) = \emptyset$$

$$Cor_E(Z_o) = \{Z_i\}$$

$$\forall a \in A_i, \quad a \in Cor_E(x) \Rightarrow type(a) = x$$

$$\forall op \in OP_i, \quad op \in Cor_E(x) \Rightarrow x \in \mathcal{A}$$

$$\forall c \in \mathcal{A}, \quad c \in Cor_E(c)$$

On suppose de plus que chaque élément de $\mathcal{A}$ possède un unique opérateur de construction, qui est d'arité nulle.

Grâce à l'extension de la correspondance, nous pouvons imposer que les opérateurs ne doivent pas donner de productions égales à une permutation près. En effet, il suffit d'utiliser les terminaux de $\mathcal{A}$ pour les distinguer le cas échéant. Cela permet d'étendre la construction des paires (t, f). Si t a pour production virtuelle p, on peut choisir pour f l'opérateur dont la production est p à une permutation près.

4.3 Utilisation de Cor_E

Une première phase de la génération de la GA de couplage est de créer, à partir des phyla et des opérateurs de G_i, les productions associées. Rappelons en effet que les algorithmes de couplage sont décrits sur des productions, et non sur des opérateurs.

Soit un phylum X et un de ses opérateurs $op \to Y_1 \dots Y_n$. Si $op \in Cor_E(C)$, alors on utilise la production $X \to C\, Y_1 \dots Y_n$. Sinon, on utilise la production $X \to Y_1 \dots Y_n$.

Ensuite, pour tout phylum Y_i tels que $\exists a, T$ et $a(Y_i) \in Cor_E(T)$, on remplace Y_i par la séquence $Y_i\, T$. La production ainsi obtenue est celle qui sera utilisée par le couplage.

Pendant la génération de la grammaire attribuée de couplage, on remplace suivant le cas toutes les références à un terminal X par:

-soit X lui-même, quand $\exists Y,\ X \in cor_E(Y)$,

-soit $a(Y)$, quand $a(Y) \in Cor_E(X)$ et que X a été inséré devant Y,

-soit l'opérateur associé à X, quand $X \in \mathcal{A}$.

On introduit ainsi deux petites modifications dans le couplage, mais qui sont très localisées, et n'interfèrent pas avec les algorithmes mis en œuvre.

La grammaire attribuée ainsi obtenue correspond exactement à ce que l'on souhaite, et l'instanciation est possible dans de nombreux cas. De plus, comme on ne modifie jamais la grammaire attribuée générique, elle peut très bien être testée en dehors de toute instanciation.

On peut imaginer d'autres méthodes que celle décrite ci-dessus, mais elles ont parfois l'inconvénient de devoir modifier le gène, ce qui met en cause sa fiabilité, et rend difficile sa mise au point et son test.

Exemple

Soit G_i une grammaire abstraite sur laquelle on souhaite instancier un gène dont la grammaire est G_g :

$$G_g = \left\{ \begin{array}{l} E = add, mul\,; \\ add \to C_1\, E\, E \\ mul \to C_2\, E\, E \end{array} \right. \qquad G_i = \left\{ \begin{array}{l} A = plus, fois\,; \\ plus \to A\, B \\ fois \to A\, B \end{array} \right.$$

où B a un attribut $a(B)$ de type A.

On peut y parvenir en spécifiant la relation de correspondance ainsi :

$$Cor_E(E) = \{A, a(B)\}$$
$$Cor_E(C_1) = \{plus\}$$
$$Cor_E(C_2) = \{fois\}$$

La grammaire G_i subit alors les modifications suivantes, avant d'être passée à l'algorithme de couplage :

$$G_i \to G'_i = \begin{cases} A = plus, fois\,; \\ plus \to C_1\, A\, B\, A \\ fois \to C_2\, A\, B\, A \end{cases}$$

Puis, B est éliminé car c'est un symbole mort, et on a deux arbres clos facilement associables aux productions de G_g. On obtient alors la grammaire attribuée suivante :

$$AG'_i = \begin{cases} where\,(plus \to C_1\, A_1\, B\, A_2)\, use \\ \ e\ := add(C_1, A_1, A_2) \\ where\,(fois \to C_2\, A_1\, B\, A_2)\, use \\ \ e\ := mul(C_2, A_1, A_2) \end{cases}$$

Que l'on transforme alors en la forme suivante (on se rappelle que A_2 a été inséré après B) :

$$AG_i = \begin{cases} where\,(plus \to A\, B)\, use \\ \quad e\ := add(C_1(), A, a(B)) \\ where\,(fois \to A\, B)\, use \\ \quad e\ := mul(C_2(), A, a(B)) \end{cases}$$

4.4 Associations sur les Arbres Clos

Le mécanisme par lequel on fabrique les paires (t, f) (cf. paragraphe 3.6) peut sembler excessivement simple. En effet, on se contente de regarder si une production virtuelle correspond exactement avec une production du gène. Il se pourrait, par exemple, que la production virtuelle soit une permutation de celle du gène. Comme on a pris la précaution de distinguer les opérateurs ayant des parties droites égales à une permutation près, on peut accepter cette extension.

On peut aussi imaginer que la production virtuelle s'identifie avec un sous-arbre de la grammaire du gène. Cette dernière extension pose en fait plus de problèmes qu'elle n'en résoud. Reconnaître des sous-arbres du gène peut produire des solutions multiples, qu'on ne peut pas discriminer sans l'intervention du programmeur. Cependant, certains sous-arbres sont triviaux, et on peut améliorer la grammaire du gène.

Dans ce qui suit, je définis une Forme Standard d'un gène, qui est une bonne réponse au problème : "Comment écire un gène pour qu'il soit instanciable sur un grand nombre de grammaires ?". En fait, quand un gène est sous cette forme, on peut classer les causes d'échec du couplage, et se rendre compte que, dans bien des cas, l'instanciation n'était pas possible.

4.5 Forme Standard

Tout au long de la description de l'algorithme de couplage, nous avons imposé des restrictions. De plus, la recherche d'une solution peut aboutir à un échec. Or certains échecs ne signifient pas que le gène n'est pas instanciable. Deux cas peuvent alors se présenter. Soit le gène n'est pas écrit sous une bonne forme, soit la correspondance ne permet pas, seule, de trouver une solution.

Dans cette section, nous allons examiner ces travers de plus près, et ainsi, nous pourrons définir une forme optimale, que nous appelterons forme standard d'un gène. Enfin, une correspondance étendue sera définie, ainsi que les modifications qu'elle impose.

La forme standard est définie par un ensemble de réductions qui transforment une grammaire attribuée sans changer le calcul qu'elle effectue. Le gène est sous forme standard quand aucune transformation ne lui est applicable.

4.5.1 [$\mathcal{R}_{AT}$] Attributs Terminaux

Comme nous l'avons vu plus haut, si des attributs aux terminaux ne sont pas définis dans les règles sémantiques, la grammaire attribuée ne peut pas être un gène sous cette forme. Plus généralement, on fait la transformation suivante :

1. Un attribut synthétisé $a(X)$ n'est jamais défini si, pour tout opérateur op où figure X, a n'est pas défini dans $F(op)$.
2. Pour un tel attribut, remplacer toutes les occurences de X par la séquence $X\,T$, où $T = type(a)$, et toutes les occurences de $a(X)$ par T.
3. Supprimer l'attribut a.

S'il reste des attributs non définis, la grammaire attribuée initiale n'était pas correcte. Donc c'est une erreur.

4.5.2 [$\mathcal{R}_{SI}$] Symboles Inutiles

Si un symbole et ses attributs n'apparaissent dans aucune règle sémantique d'un opérateur op, alors ce symbole n'est pas utile aux calculs. Or il figure dans la production associée à op. Donc, si une grammaire ne contient pas ce symbole, on ne pourra pas faire l'instanciation, alors que le calcul peut se faire. Il faut donc l'éliminer de op.

4.5.3 [$\mathcal{R}_{AP}$] Applatissement du Gène

Si on assimile un non-terminal à un type, une production peut alors être vue comme une structure (C). Or, ce qui est important d'un point de vue algorithmique, ce sont les récursions. Ainsi, quand on a la grammaire suivante :

$$G_1 : \left\{ \begin{array}{l} X \rightarrow AY \\ Y \rightarrow BX \end{array} \right.$$

On observe que, finalement, X regroupe trois champs : A, B et X. Si on veut instancier un algorithme définit sur G_1 sur la grammaire suivante :

$$G_2 : \left\{ X \rightarrow ABX \right.$$

L'algorithme de couplage va échouer, car il ne trouvera pas de correspondance pour la sous-structure Y. Pourtant, la structure récursive des données est la même. D'où la transformation $\mathcal{R}_{AP}$, sur une grammaire attribuée AG dont la grammaire est G :

1. On définit la relation $Y \underset{X}{\overset{*}{\Rightarrow}} Z$ ssi on a

$$\left\{ \begin{array}{l} Y \Rightarrow Y_1 \Rightarrow \ldots Y_n \Rightarrow Z \\ \forall i, Y_i \neq X \\ Y \neq X \end{array} \right.$$

2. Pour un opérateur $p : X \rightarrow X_1 \ldots X_n$, R_p est le plus grand ensemble inclus dans $\{X_1 \ldots X_n\}$ tel que :

$$X_i \underset{X}{\overset{*}{\Rightarrow}} X_i \Rightarrow X_i \notin R_p$$

3. Pour chaque $X_i \in R_p$ et pour chaque opérateur f de X_i, créer l'opérateur $p.f$ de X tel que :

$$\begin{array}{rl} f & : X_i \rightarrow X_i^1 \ldots X_i^k \\ p.f & : X \rightarrow X_1 ... X_{i-1}, X_i^1 ... X_i^k, X_{i+1} ... X_n \end{array}$$

4. On adapte les règles sémantiques en utilisant la méta-composition. On définit une grammaire attribuée AG_c de couplage qui est l'identité sur G et telle que pour $p.f$ on synthétise le sous arbre :

$$p(X_1 ... X_{i-1}, f(X_i^1 ... X_i^k), X_{i+1} ... X_n)$$

On a alors $\mathcal{R}_{AP}(AG) = AG \otimes AG_c$.

On remarque qu'un terminal ayant encore un attribut est automatiquement éliminé par cette transformation (on est dans le cas où $k = 0$).

4.5.4 Réduction sous Forme Standard

On dit qu'une grammaire est sous forme standard lorsqu'aucune transformation parmi $\mathcal{R}_{AT}$, $\mathcal{R}_{SI}$ et $\mathcal{R}_{AP}$ n'est possible. La terminaison de cette réduction est assurée par la démonstration suivante. On a :

-$\mathcal{R}_{AT}$ réduit strictement le nombre d'attributs, mais augmente le nombre de symboles.

-$\mathcal{R}_{SI}$ n'augmente jamais le nombre d'attributs ou de symboles.

-$\mathcal{R}_{AP}$ diminue strictement le nombre de symboles, et n'augmente pas le nombre d'attribut.

Supposons qu'il existe une suite infinie $f = (f_1, f_2, \dots)$ de réductions d'une grammaire attribuée. Si $at = \max\{i \mid f_i = \mathcal{R}_{AT}\}$, on a forcément $at < +\infty$, sinon le nombre d'attributs deviendrait négatif.

Donc, pour $j > at$, f_j n'augmente plus le nombre de symboles. Donc, si $ap = \max\{j > at \mid f_j = \mathcal{R}_{AP}\}$ on a forcément $ap < +\infty$, sinon le nombre de symboles deviendrait négatif.

Donc pour $k > at$, on a $f_k = \mathcal{R}_{SI}$. Or, si on applique successivement une infinité de fois cette réduction, un opérateur deviendrait d'arité négative.

Donc, f n'existe pas et la réduction sous forme standard se termine.

4.5.5 Productions dans une Forme Standard

On peut maintenant caractériser la forme générale des productions dans une grammaire sous forme standard. On peut en fait distinguer trois genres de structures différentes :

- Les formes non récursives :

  $$X \rightarrow A_1 \dots A_n$$

  où les A_i sont des terminaux, ou des objets récursifs qui ne dérivent pas dans X. Ces structures sont les types enregistrements classiques des langages de programmation.

- Les formes de listes :

  $$X \rightarrow A_1 \dots A_n\, X$$

  qui sont également très courantes.

- Les différentes formes d'arbres, qui ne sont jamais que des listes imbriquées :

  $$X \rightarrow A_1 \dots A_n\, Y_1 \dots Y_p, X \dots X$$

  où les Y_i sont des structures mutuellement récursives avec X.

Les transformations définies plus haut garantissent que l'on a pas d'autres types de structures. Ce résultat n'est pas très intéressant en soit, mais il permet d'affirmer que si l'algorithme de couplage échoue, c'est parce qu'il manque dans la correspondance une structure récursive fondementale pour le gène.

4.6 Forme Standard et Comparaison

Déterminer si deux algorithmes font le même calcul est un problème malheureusement indécidable. Si on cherche à définir une égalité entre deux grammaires attribuées, on est donc réduit à comparer leurs structures. Or la réduction sous forme standard a l'avantage de dégager ce qui est crucial dans la structure des données manipulées, et d'éliminer ce qui est inutilisé dans les calculs.

On peut donc définir l'égalité de deux grammaires attribuées en comparant si leurs formes standards sont égales à un renommage près.

Par rapport à l'opération de couplage, on peut vouloir comparer deux gènes en fonction du nombre de grammaires sur lesquelles ils peuvent être instanciés. De ce point de vue, la forme standard d'un gène est strictement meilleure, car on a vu que pour chaque transformation, de nouvelles grammaires peuvent être instanciées.

Ces éléments amènent à penser que la forme standard d'une grammaire attribuée est une bonne caractérisation de ce qu'elle fait, tout en étant fortement réutilisable.

A la question "comment écrire un gène ?" nous pouvons donc répondre que la forme standard du gène est la plus efficace en termes de possibilités d'instanciation par l'algorithme de couplage.

Chapitre 5

Implémentation

L'algorithme de couplage à été implanté dans le système Fnc-2, ainsi que les extensions de la correspondance étendue. L'application est entièrement écrite en OLGA, dont la syntaxe est suffisamment parlante. Les listings complets sont reportés en annexe, ainsi que les differents tests.

5.1 Description de l'Application

Pour mémoire, rappelons que l'application Couplage ne produit que la grammaire attribuée de couplage à partir de la relation de correspondance, d'un gène, et d'une grammaire. L'instanciation proprement dite est effectuée, de manière externe, en utilisant l'opérateur de méta-composition du système Fnc-2.

La structure générale de l'application est donnée figure 5.1, et ne présente en fait aucune particularité par rapport au schéma standard de la figure 2.1.

En fait, l'application se compose de différents modules, correspondant aux différentes étapes de l'algorithme de couplage. Nous avons dans l'ordre :

1. Un "parser" du texte source contenant la description de la relation de correspondance. Ce module est créé par Syntax à partir d'une grammaire concrète de type BNF.

2. Un "type-checker" qui teste si les contraintes sur la relation de correspondance sont vérifiées.

3. Un module de transformation de la grammaire G_i sur laquelle doit s'instancier le gène G_o. C'est notamment ici que la correspondance étendue est utilisée.

4. Un module de calcul des arbres clos, et de création des paires (t, f).

5. Un module de recherche de solutions et d'impression.

Tous ces modules sont reliés entre eux par une grammaire attribuée, qui gère le flot de données.

L'application prend en entrée une relation de correpondance écrite dans un mini-langage adapté, et produit en sortie le texte source en OLGA de la

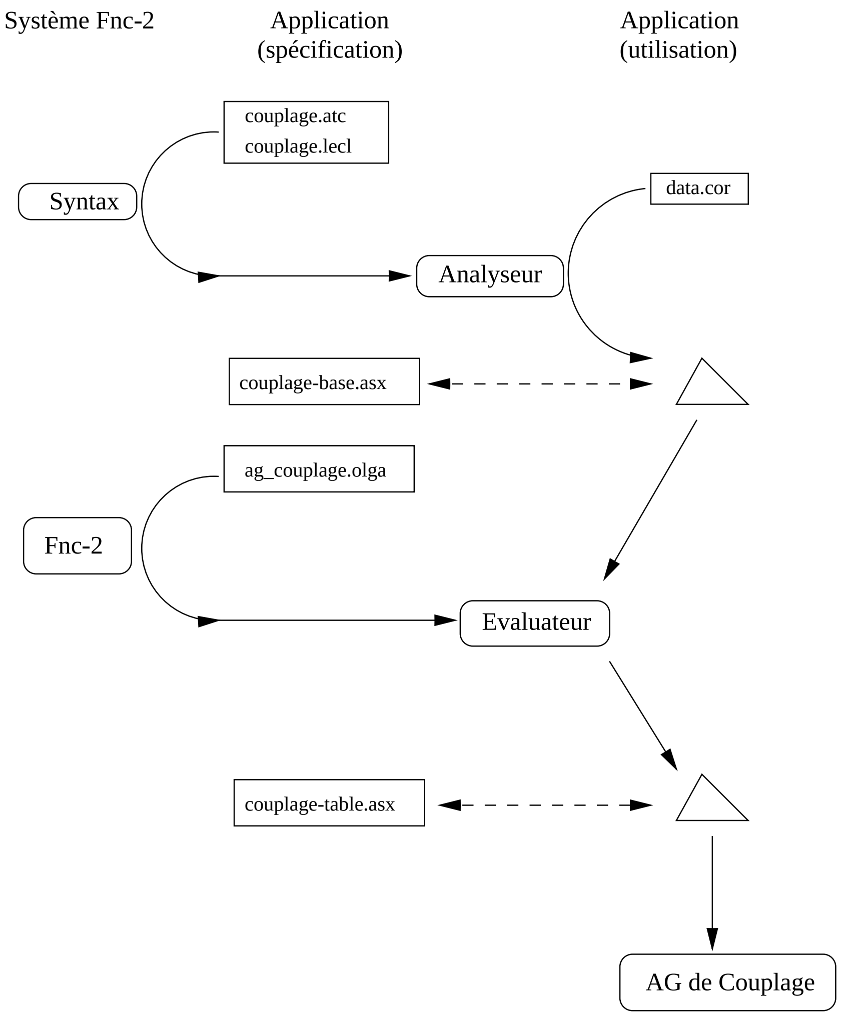

FIG. 5.1 – *Application Couplage*

grammaire attribuée de couplage. L'application se charge d'aller chercher sur le disque les informations dont elle a besoin à propos des deux grammaires G_i et G_o, dont les noms figurent dans l'entête de la relation de correspondance.

5.2 Syntaxe d'Entrée

La relation de correspondance est spécifiée dans un mini-language dont la syntaxe (BNF) est illustrée figure 5.2.

```
<cor prog> =    <header>
                <cor+>
                end cor“;”

<header>=       cor <ID> from <ID> to <ID> “;”

<cor+>=         <cor def> <cor+>
<cor+>=         <cor def>

<cor def>=      <ID> “=” <cor img*> “;”

<cor img*>=     <cor img+>
<cor img*>=     .
<cor img+>=     <cor img> “,” <cor img+>
<cor img+>=     <cor img>

<cor img>=      <ID>
<cor img>=      “$” <ID> “(” <ID> “)”
```

FIG. 5.2 – *Syntaxe concrète de Couplage*

Les fichiers *couplage.atc*, *couplage.lecl*, *couplage.prio* et *couplage.recor* sont destinés à Syntax, et contiennent respectivement : la syntaxe BNF, la description des éléments lexicaux (ID, commentaires…), les priorités, et la récupération des erreurs.

```
CORPROG → HEADER CORL
HEADER  → ID ID ID
CORL    → COR+
COR     → ID IMG
IMG     → IDimg*
```

FIG. 5.3 – *Syntaxe abstraite de Couplage*

A partir de ces fichiers, Syntax produit un analyseur syntaxique qui donne en sortie un arbre abstrait. La grammaire abstraite pour construire cet arbre est donnée figure 5.3.

5.3 Type-Checker

Ce vérificateur se contente de vérifier trois propriétés. Premièrement, que tous les phylums mis en correspondance sont bien dans les grammaires utilisées. Deuxièmement, qu'un phylum n'est pas mis en correspondance avec deux phyla du gène, et qu'il n'y a pas de double définition. Enfin, que les racines sont mises en correspondance.

Pour cette vérification, on utilise une grammaire attribuée dont les attributs sont les suivants :

- tn_o(CORL,COR) : liste des phyla du gène (G_o)
- tn_i(CORL,COR,IMG,IDimg) : liste des phyla de la grammaire recevant le gène (G_i).

Ces attributs sont hérités, et leur initialisation se fait dans HEADER, en allant chercher sur le disque les informations des grammaires utilisées. En fait HEADER possèdent deux attributs synthétisés stni et stno, dont les valeurs sont passées à tno et tni par CORPROG.

- s_used_tno & h_used_tno (CORL,COR) : liste des phyla de G_o utilisés, respectivement après et avant la visite du nœud courant.
- s_used_tni & h_used_tni (CORL,COR,IMG,IDimg) : liste des phyla de G_i utilisés, respectivement après et avant la visite du nœud courant.

Ces attributs permettent de faire les vérifications attendues. Si en effet dans un nœud du type COR, on a met en correspondance un phylum qui existe dans h_used_tni(COR), alors ce phylum est déjà en correspondance avec quelque chose, donc il y a une erreur. Dans le cas contraire, s_used_tni(COR) est égal à h_used_tni(COR) plus le phylum mis en correspondance.

Trois attributs sont utilisés pour transmettre les erreurs :

- correct(CORPROG,HEADER,CORL,COR,IMG,IDimg) : indique si la correspondance est correcte pour le nœud courant.
- rootok(CORL,COR) : indique si la racine à été définie au moins une fois.
- kind(IMG,IDimg) : indique si on est en train de mettre en correspondance la racine.

Pour la description de la grammaire attribuée, voir les listings en annexe. La grammaire attribuée est dans le fichier “ag_couplage.olga”.

5.4 Le module “couplage_types.olga”

Ce module décrit les types élémentaires utilisés dans l’application. Notamment le type ident_tn, qui est tout simplement un identificateur suivit d’un type (phylum, operateur, clef, neutre, etc.)

Le module contient les fonctions pour aller chercher la description des grammaires sur le disque, ainsi que certaines opérations sur les listes d’identificateurs.

5.5 Le module “couplage_key_func.olga”

On trouve dans ce module les opérations sur les attributs, et sur les relations. Ce module permet de faire certains calculs élémentaires, comme la fermeture transitive d’une relation. C’est également ici que sont calculées les relations $\stackrel{*}{\Rightarrow}$ et $\underset{Cor}{\stackrel{*}{\Rightarrow}}$, et la liste des symboles clefs et neutres.

Enfin, et surtout, ce module calcule la liste des symboles neutres et récursif, et détermine les classes d’équivalence de la relation Γ définie en 3.5.

5.6 Le module “couplage_closed.olga”

Ce module calcule tout simplement les arbres clos, en supposant que le cas des neutres récursifs ait été traîté. C’est une implémentation directe de l’algorithme 3.2.

5.7 Le module “couplage_link”

Ce module fait l’association entre arbre clos et opérateurs de construction d’arbre. Il construit en fait les paires (t, f). Le mécanisme implémenté compare les productions virtuelles aux opérateurs du gène, sans permutation, et sans chercher de sous-arbres. Néanmoins, le programme a été écrit pour permettre ces extensions. Par exemple, les opérateurs de constructions d’arbres sont décris par le type TCF, qui permet de décrire des termes finis du premier ordre. Avec les deux opérateurs p et q, on peut donc définir le constructeur $op(x, y, z) = p(x, q(y, z))$.

5.8 Le module “couplage_AG.olga”

C’est le module final, le plus important et aussi le plus long. On peut y distinguer trois parties internes.

La première est chargée de l’impression de la grammaire attribuée. Cette partie est plus complexe qu’un simple parcours des structures du programme, car la grammaire attribuée est stockée sous forme “compactée”. En effet, les neutres récursifs ont été renommés par classe d’équivalence, et les attributs sont rangés par phylum et non par attribut.

La deuxième partie est une implémentation des algorithmes 3.5 et 3.6, pour la génération des AG partielles.

La dernière partie gère l'exploration des différentes solutions, en implémentant l'algorithme 3.4.

5.9 Tests

Je présente ici trois tests représentatifs de l'application Couplage. Dans ces exemples, il n'y pas de description de l'algorithme instancié, qui ne concerne que la phase finale utilisant la méta-composition. Seule la grammaire attribuée de couplage des deux grammaires (G_{in} et G_{out}) sont donc présentées.

La sortie de l'application est restitué tel quel. Le fichier ainsi généré est écris en OLGA, mais la partie utile est entourée de nombreuses lignes de commentaires fournies lors des différents calculs intermédiaires. Ces commentaires sont placés entre accolades. Ainsi, on trouve :

- CORET : grammaire d'entrée après l'action de la correspondance étendue.
- INITIAL : ensemble des sous-arbres initial pour le calcul des arbres clos. Les opérateurs qui étiquettent les nœuds internes sont donnés entre crochets.
- trace de l'exploration : indications des actions effectuées pendant l'exploration des solutions. On peut notemment y suivre les profondeurs courantes et maximales.
- Programme en OLGA : il commence par "attribute grammar..." et se termine par "end grammar ;".
- TERM : liste des terminaux du gène.
- INSTANCES : liste des paires (t, f). Les fonctions sont données par leur prototype.
- CLOSED : liste des arbres clos.
- CLASSES : classes d'équivalences de la relation Γ.
- NewGi : grammaire obtenue après élimination des récursions et des symboles morts.
- NEU : liste des symboles neutres.
- KEY : liste des symboles clefs.
- Gout : grammaire du gène.
- Gin : grammaire d'entrée.

L'apparent désordre de cette liste vient du non-déterminisme de l'ordre dans lequel sont évalués les attributs de la grammaire attribuée de l'application. De plus, comme ces affichages sont produit par effet de bord, il ne reflettent pas toujours l'ordre dans lequel sont effectuées les calculs.

5.9.1 Exemple 1

Cet exemple met en valeur la recherche de solution compatibles. On peut avoir en effet que deux constructions sont possibles pour le non-terminal Y, mais une seule est compatible avec la construction en X.

On voit d'ailleur que pendant l'exploration, la profondeur maximale a atteint zéro, signe d'un changement de décision.

5.9.2 Exemple 2

Cet exemple montre l'élimination des récursions sur les non-terminaux neutres. Dans la grammaire attribuée de couplage, on ne doit pas s'inquiéter de l'absence de règles pour l'opérateur c. En effet, les règles de copies entre éléments d'une même classe déquivalence sont du type $a(D_0) := a(D_j)$, et sont donc automatiquement crées par Fnc-2.

5.9.3 Exemple 3

Cet exemple est un peu plus compliqué, mais plus réaliste. On cherche a instancier sur un langage fonctionnel simplifié l'utilisation d'une table de symbole.

L'abstraction qui est faite dela table de symbole est la suivante : Un programme est une liste de déclarations et d'instructions. Hélas, comme l'application de couplage de recherche pas de sous-arbres de hauteur supérieure à un, cette simple déclaration ne suffit pas, et il faut donc utiliser un gène sous forme standard. De plus, il faut ajouter quelques cas particuliers de sous-arbres de hauteur 2 ou 3. Mais la structure fondammentale reste inchangée.

Conclusion

Ce travail sur la généricité est la suite logique des idées de [GG84] et [FMY92]. Dans ces papiers, seule la notion de couplage était étudiée, mais il n'était pas question de générer les grammaires attribuées de couplage par un algorithme. De plus, l'utilisation du couplage pour obtenir de la généricité n'était qu'évoquée. Avec la méta-composition et les premièrs algorithmes de couplage [BJPR93], Didier Parigot et Martin Jourdan ont créé un outil fort utile, intermédiaire entre le polymorphisme et l'héritage d'objets. La possibilité d'instancier, en les spécialisant, des algorithmes optimisés est très interessante, tant du point de vue du temps de dévelloppement que de la robustesse de l'application crée.

Avec cette première implémentation de l'algorithme de couplage, nous allons pouvoir expérimenter le réel pouvoir d'expression de cette forme de généricité. Avec la définition de la forme standard, nous avons déjà franchi un cap dans cette expérimentation. Non seulement cette forme donne une manière d'écrire les gènes, mais elle donne les causes d'échecs de l'instanciation, et permet de les caractériser. On pourra ainsi orienter les extensions de l'algorithme de couplage afin de n'envisager que celles qui seront véritablement utiles.

Enfin, ce travail sur la généricité, en parallèle avec des travaux sur la comparaison entre languages fonctionnels et grammaires attribuées, donne des perspectives. Pourquoi ne pas définir des outils de généricité dans d'autre paradigmes de programmation ? Les grammaires attribuées, utilisées en tant que représentation intermédiaire d'un programme, seraient alors un bon outil pour définir ces "généricités" et effectuer les instanciations.

Bibliographie

[BD88] Pierre Boullier and Philippe Deschamp. *Le SystÈMe SYNTAX— Manuel D'Utilisation Et De Mise En OEUvre Sous Unix.* —INRIA—, September 1988.

[BJPR93] 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, 1993. Springer-Verlag.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science.* Springer-Verlag, New York–Heidelberg–Berlin, August 1988.

[FMY92] 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 Progr. Languages*, pages 223–234. Albuquerque, NM, January 1992.

[GG84] 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, volume 19, number 6.

[JLP90] Martin Jourdan, Carole Le Bellec, and Didier Parigot. The olga attribute grammar description language: Design, implementation and evaluation. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 222–237. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

[JP88] Martin Jourdan and Didier Parigot. More on speeding up circularity tests for attribute grammars. rapport RR-828, INRIA, Rocquencourt, April 1988.

[JP89] Martin Jourdan and Didier Parigot. The FNC-2 system: Advances in attribute grammar technology. In O. M. Tammepuu, editor, *Procs. of* *the Soviet-French Symposium Informatika '89*, pages 94–118, Tallinn, May 1989. See also: rapport RR-834, INRIA, Rocquencourt (April 1988).

[JP90] Catherine Julié and Didier Parigot. Space optimization in the FNC-2 attribute grammar system. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 29–45. Springer-Verlag, New York–Heidelberg–Berlin, September 1990. Paris.

[JP91] Martin Jourdan and Didier Parigot. Internals and externals of the fnc-2 attribute grammar system. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 485–504. Springer-Verlag, New York–Heidelberg–Berlin, June 1991. Prague.

[Knu68] 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).

[Rou92] Gilles Roussel. Élimination de suites de règles de copies dans les grammaires attribuées. Rapport de Recherche à paraître, INRIA, Rocquencourt, 1992.

[Rou94] 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.