Table des matières

Table des figures

Chapitre 1

Introduction

Depuis les débuts de l'informatique, les langages de programmation ont beaucoup évolué. Au commencement le programmeur parlait en binaire ; puis comme il trouvait cette façon de programmer un peu "terre à terre", il a voulu s'éloigner de la machine en utilisant des langages de plus haut niveau. Ainsi sont nés des langages comme PL/1, Algol 60,.... Par la suite de nombreux langages ont été créés comme Prolog, CAML,..., chacun introduisant de nouveaux concepts, fruits d'études théoriques ou d'expérimentations.

La distance entre les langages de programmation et le langage binaire parlé par les machines est donc de plus en plus importante. Les traducteurs ou compilateurs de ces langages de haut niveau vers un langage machine sont donc de plus en plus complexes. Si, pour les premiers langages, la construction des traducteurs pouvait être empirique, avec la complexification du processus de traduction, il a fallu introduire des bases plus formelles.

Les premiers apports concernent la théorie des langages introduite par Chomsky et Schutzenberger. Elle permet de décrire de façon rigoureuse et simple les langages de programmation pour leurs aspects lexico-syntaxiques non-contextuels. Grâce à la théorie des automates couplée avec la théorie des langages, il est ensuite possible de construire un analyseur lexico-syntaxique à partir de la description du langage. Ces analyseurs ont une efficacité équivalente à celle des analyseurs écrits à la main.

Une fois l'aspect lexico-syntaxique du traducteur décrit, il est nécessaire de lui attacher une sémantique. Là aussi de nombreux travaux théoriques ont été effectués et plusieurs voies ont été explorées comme la sémantique algébrique, la sémantique dénotationnelle, la sémantique opérationnelle ou les types abstraits algébriques.

D'autres méthodes s'intéressent aux aspects sémantiques qui dépendent du contexte. On dit qu'elles traitent la sémantique statique. Cette approche se retrouve dans différentes méthodes de description comme les doubles grammaires, les grammaires affixes ou les **grammaires attribuées**.

Malgré leur puissance formelle, ces méthodes ne permettent pas, en général, de construire automatiquement des compilateurs efficaces en temps de traitement et en espace mémoire (comparés à un compilateur écrit à la main). Seules les grammaires attribuées peuvent répondre à cette attente, dans la mesure où l'on adopte une solution efficace pour les exécuter. Une liste des différentes solutions proposées et de différents systèmes existants est donnée dans [1]. C'est sur cette méthode de spécification de la sémantique statique que porte ce mémoire.

Cette méthode a été introduite par [2] et nous allons ici en redonner les points importants. Étant donnée une grammaire indépendante du contexte, à chacun des non-terminaux de la grammaire le programmeur associe un ensemble d'attributs qui sont des symboles qui peuvent contenir des valeurs. Certains attributs permettent de véhiculer l'information des feuilles d'un arbre vers sa racine, ils sont appelés attributs synthétisés ; d'autres permettent de transporter l'information en sens inverse, ils sont appelés attributs hérités. Sur chaque production, le programmeur spécifie ensuite comment calculer ces attributs grâce à des règles sémantiques.

Le processus est alors le suivant. A partir d'un texte l'analyseur lexico-syntaxique construit un arbre. Conformément à la spécification de la grammaire attribuée, des attributs sont associés aux différents nœuds de l'arbre. Un évaluateur d'attributs est alors chargé de calculer l'ensemble des attributs que l'on trouve sur l'arbre, conformément aux règles sémantiques.

Cette présentation rapide nous montre que les grammaires attribuées permettent de décrire facilement des compilateurs. Cette méthode de description a l'avantage d'être déclarative : la spécification décrit *ce* qu'il faut calculer, mais pas *comment* cela va être calculé. En outre, elle est intrinsèquement structurée par l'ensemble des productions : pour chaque production le programmeur spécifie un ensemble de règles sémantiques.

Afin de produire des compilateurs de bonne qualité, il est nécessaire de trouver une méthode d'évaluation des attributs d'un arbre qui soit efficace du point de vue du temps de calcul et de la place mémoire. Les différents travaux théoriques [1,3] ont permis de décrire un ensemble de classes de grammaires attribuées pour lesquelles il est facile de construire des évaluateurs efficaces. Pour chacune de ces classes, ce qui est perdu en pouvoir d'expression permet de gagner en efficacité des évaluateurs produits.

Dans le chapitre 2 de ce mémoire, nous rappelons les différents résultats théoriques et les classes les plus intéressantes de grammaires attribuées (non-circulaire, FNC, DNC et l-ordonnée). Nous revenons rapidement dans les chapitres 4 et 6 sur la notion de classes de grammaires attribuées, en introduisant de nouvelles classes. Mis à part la classe DNC′ qui est une variante plus large de la classe DNC que nous introduisons dans le chapitre 2, les nouvelles classes ne sont pas des classes génériques comme celles données dans le chapitre 2. Elles s'inspirent de la forme de la spécification de la grammaire attribuée pour construire une classe mieux adaptée à celle-ci ; il est alors possible de construire des évaluateurs de bonne qualité pour des grammaires attribuées qui classiquement ne pouvaient pas être traitées efficacement.

Le but premier de ce mémoire n'est pas de décrire de nouvelles classes de grammaires attribuées, mais d'essayer d'introduire les notions de modularité et de réutilisabilité dans la méthode de spécification par grammaires attribuées.

En effet, même s'il est possible de voir l'ensemble des règles sémantiques associées à une production comme un petit module dans lesquel est décrit un ensemble de calculs relatifs à la production, il n'existe pas de véritable possibilité d'écriture modulaire comme cela est possible dans la plupart des langages de programmation, même les plus anciens. Le programmeur se retrouve donc souvent en face d'une énorme grammaire attribuée décrivant la totalité de son application ; la grammaire attribuée qui décrit le front-end ADA correspond à un livre de 500 pages. Ces grammaires attribuées sont très difficiles à déchiffrer du fait du nombre très important d'attributs et de règles sémantiques attachées à chaque production. Il paraît donc nécessaire de pouvoir décrire une application en éclatant sa spécification dans différents modules résolvant chacun une partie **complète** du problème. Ces modules doivent, lorsqu'ils sont réunis, être équivalents à la spécification complète du problème mais il faut également que cette décomposition ne soit pas pénalisante au niveau de l'efficacité. Les grammaires attribuées telles que nous les avons présentées jusqu'à présent ne permettent pas une telle modularité.

C'est la raison pour laquelle les grammaires couplées par attributs, ou grammaires attribuées fonctionnelles, ont été introduites dans [4]. Elles permettent de décomposer un problème en phases, chacune construisant à partir d'un arbre d'entrée un nouvel arbre qui est l'arbre d'entrée de la phase suivante. Le passage par un arbre intermédiaire permet de rendre la spécification d'un problème plus simple dans la mesure où la syntaxe de ce nouvel arbre s'adapte mieux à l'expression du problème. En outre, les auteurs ont introduit un algorithme qui permet, à partir d'une séquence de grammaires attribuées fonctionnelles, de construire une grammaire attribuée ayant la même sémantique. Cette algorithme est appelé composition descriptionnelle ou méta-composition. Il a l'avantage de ne pas construire les arbres intermédiaires comme l'aurait fait la séquence de grammaires attribuées. Grâce à cet algorithme, il est possible de concilier écriture modulaire et évaluation efficace.

Une première partie de ce mémoire (chapitre 3) porte sur les grammaires attribuées fonctionnelles et leur méta-composition. La méta-composition paraît particulièrement intéressante dans le cadre du système de traitement des grammaires attribuées FNC-2 développé dans le projet “ChLoE” à l'INRIA Rocquencourt, car celui-ci permet déjà d'écrire des grammaires attribuées fonctionnelles.

Malheureusement les résultats théoriques ne s'appliquent pas directement dans le cadre d'un système réel, notamment à cause du traitement incomplet des conditionnelles et du langage très restreint sur lequel est décrite la méta-composition. Des restrictions importantes sur la classe des grammaires attribuées acceptées en entrée de la méta-composition diminuent encore l'applicabilité de l'algorithme théorique.

Nous proposons donc ici différentes extensions à l'algorithme de base qui permettent d'augmenter la puissance d'expression de la méta-composition et d'élargir la classe des grammaires attribuées acceptées par l'algorithme. Ensuite, nous décrivons un algorithme qui permet un meilleur traitement des conditionnelles “*If then else*”.

Un deuxième point faible de la méta-composition, dans le cadre d'une application pratique, concerne l'efficacité des grammaires attribuées produites. En effet, celles-ci peuvent contenir des règles de copie qui, sur tout arbre, transportent des attributs du haut vers le bas puis du bas vers le haut d'une façon complètement inutile car non productives.

Nous proposons donc un ensemble d'algorithmes qui, à partir de l'analyse statique d'une spécification, détectent ces suites de règles de copie et transforment cette spécification afin de les faire disparaître. Les grammaires attribuées ainsi transformées permettent une évaluation plus efficace. Il faut noter que ces algorithmes peuvent être appliqués indépendamment de la méta-composition.

Toutes ces extensions ont servi de base à une véritable implantation de la méta-composition au sein du système de traitement des grammaires attribuées FNC-2. C'est la seule implantation complète de l'algorithme original. D'autres modifications de la méta-composition ont été nécessaires afin d'accepter les spécificités du langage d'entrée de FNC-2.

Malheureusement, nous découvrons les limites de cet algorithme : non clôture des classes de grammaires attribuées vis-à-vis de la méta-composition et impossibilité de faire de la compilation séparée. Cette dernière caractéristique semble être un obstacle majeur à l'utilisation de la méta-composition dans le cadre d'une application importante [5]. Chaque fois qu'un module est modifié, il est nécessaire de reconstruire complètement la grammaire attribuée ayant la même sémantique que la séquence de modules, ainsi que l'évaluateur associé. De plus, la classe des grammaires attribuées FNC n'est pas close pour la méta-composition. Il n'est donc pas toujours possible de construire un évaluateur FNC, même si nous sommes capables de construire un évaluateur FNC pour chacun des modules.

Pour répondre à cette attente de compilation séparée, nous avons introduit une nouvelle approche, la *composition des évaluateurs*. Celle-ci permet, tout en gardant les bonnes propriétés de la méta-composition, de pouvoir construire des évaluateurs efficaces quels que soient les modules constituant la séquence. Elle est fondée sur la même idée que la méta-composition, mais est appliquée, non plus au niveau des spécifications mais au niveau des évaluateurs. Nous avons ensuite introduit la notion de *filtre de couplage* afin de permettre une véritable compilation séparée.

Tout comme pour la méta-composition, de nombreuses extensions viennent modifier l'algorithme de base, afin d'élargir la classe des grammaires attribuées acceptées et d'éliminer les règles de copie inutiles.

La modularité introduite par les grammaires attribuées fonctionnelles n'est pas suffisante pour de nombreux utilisateurs des grammaires attribuées. Une décomposition en différents modules contenant chacun la résolution d'un sous-problème particulier semble mieux adaptée à certains problèmes. Un certain nombre de travaux [5–7] se sont intéressés à ce type de modularité. Notre approche est comparable à celle de Farrow mais diffère en particulier dans le mode d'évaluation des modules constituant une application.

Une troisième partie de ce mémoire concerne donc l'introduction des *grammaires attribuées incomplètes* qui permettent une telle écriture modulaire. Les grammaires attribuées sont décomposées au niveau de la sémantique, chaque module travaillant sur une même syntaxe. Nous proposons d'abord des extensions des algorithmes de détermination des classes existants afin de tester ces modules séparément ; nous avons ensuite une démarche comparable à celle utilisée pour les grammaires attribuées fonctionnelles. Premièrement, nous introduisons la *méta-union* qui, à partir de l'ensemble des spécifications, produit une unique grammaire attribuée ; nous en décrivons certaines propriétés mais, tout comme pour la méta-composition, nous en voyons les limites. Deuxièmement, nous introduisons une compilation séparée des différents modules qui permet d'obtenir des évaluateurs efficaces.

Nous avons dans cette première partie mis en place différentes techniques permettant une écriture modulaire des grammaires attribuées afin de rendre la programmation par grammaires attribuées plus attrayante. L'étape suivante pour permettre une écriture plus "agréable" des grammaires attribuées consiste à pouvoir réutiliser certains morceaux de grammaires attribuées. Modularité et réutilisabilité sont donc deux notions fortement liées. De plus, il est plus facile d'utiliser dans une grammaire attribuée un morceau d'une grammaire attribuée déjà écrite quand celui-ci est séparé du reste de l'application sous forme d'un module.

Le problème de réutilisabilité n'est pas un problème trivial comme dans d'autres langages. En effet la spécification par grammaire attribuée est très liée à la syntaxe et, si la syntaxe change un peu, la spécification s'en trouve complètement modifiée. Pourtant, on remarque que certains problèmes tels que la création de table des symboles se retrouvent dans différentes grammaires attribuées. Plusieurs approches ont déjà été proposées [5, 8] mais nous trouvions que celles-ci laissaient encore trop de travail à l'utilisateur. Nous avons donc recherché un processus le plus automatique possible. Notre travail est fondé sur la notion de *grammaires attribuées génériques*, ou gènes, introduites dans [9], grammaires attribuées qui résolvent sur une syntaxe minimale un problème donné.

Plusieurs stratégies d'instanciation du gène sur une syntaxe réelle sont alors possibles ; nous avons choisi la stratégie que nous pensons la plus automatique possible. Nous laissons l'utilisateur associer les non-terminaux et terminaux de la syntaxe réelle à ceux de la syntaxe du gène, et nous essayons ensuite de construire automatiquement une grammaire attribuée fonctionnelle passant de l'une à l'autre. Il suffit ensuite d'appliquer la méta-composition pour obtenir une grammaire attribuée ayant la même sémantique que le gène. Nous décrivons dans le chapitre 7 un algorithme très simple de construction de la grammaire attribuée fonctionnelle de passage et de petites extensions.

Malheureusement nous ne disposons pas encore d'une bibliothèque complète de grammaires attribuées génériques nous permettant de contruire une application à partir de différents gènes. Pourtant les nombreuses lignes de code déjà écrites contiennent la résolution de nombreux problèmes classiques. Il paraît donc très intéressant de pouvoir extraire de grammaires attribuées déjà écrites ces solutions et les transformer afin qu'elles puissent être utilisées comme des gènes. Nous proposons donc un algorithme simple d'extraction automatique de gène à partir d'une grammaire attribuée déjà écrite.

Les différentes techniques présentées dans ce mémoire ont toutes un but commun : rendre la programmation par grammaires attribuées plus attrayante. Pour cela nous proposons des solutions aux problèmes de modularité et de réutilisabilté dans les grammaires attribuées. De plus, afin que l'utilsateur ne souffre pas au niveau de l'efficacité de ce mode de programmation, nous proposons différents algorithmes de transformation et d'optimisation.

Chapitre 2

Généralités sur les grammaires attribuées

2.1 Définitions et Notations

Dans cette section nous introduisons un ensemble de définitions et de notations pour décrire de façon formelle la notion de grammaire attribuée. Nous présentons également de façon abstraite l'idée d'évaluation d'une grammaire attribuée sur un arbre d'entrée. Toutes les notations présentées ici seront reprises dans l'ensemble de ce mémoire.

Grammaires Attribuées

Bien que l'objet de ce mémoire ne soit pas l'analyse syntaxique et que la notion importante pour les grammaires attribuées ne soit pas le texte source mais l'arbre syntaxique, la spécification par grammaire attribuée est fortement liée, comme son nom l'indique, à la notion de grammaire. C'est pourquoi nous introduisons ici de façon formelle la définition d'une grammaire indépendante du contexte.

Définition 1 (Grammaire Indépendante du Contexte) *On appelle grammaire indépendante du contexte (CF-grammaire) le n-uplet* $(N, T, P, \mathcal{Z})$ *avec :*

- N *un ensemble de non-terminaux ;*
- T *un ensemble de terminaux,* $N \cap T = \emptyset$ *;*
- P *un ensemble de productions,* $P \subset N \times [N \cup T]^*$ *;*
- $\mathcal{Z}$ *l'axiome,* $\mathcal{Z} \in N$*; il n'apparaît jamais en partie droite d'une production.*

Nous supposons qu'il existe un moyen d'analyser le langage engendré par G, $L(G)$ et de construire les arbres syntaxiques correspondants.

Pour garder une certaine cohérence entre les différentes parties de ce mémoire, nous proposons dès maintenant de voir les non-terminaux comme des types syntaxiques et les terminaux comme des types sémantiques. Nous permettons aux terminaux d'être des terminaux génériques. Nous supposons que lors de l'analyse d'une phrase, il est possible de récupérer la valeur des terminaux, conformément à leur type sémantique associé.

A chaque production de P de la forme :

$$(\mathcal{X}_0, \mathcal{X}_1, \cdots, \mathcal{X}_n)$$

où $n \geq 0, \mathcal{X}_0 \in N$ et $\mathcal{X}_i \in N \cup T, 1 \leq i \leq n$,
nous associons une production nommée de la forme :

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

avec $i \neq j \Rightarrow X_i \neq X_j$.

p est le nom unique de la production.
Si $\mathcal{X}_i \in N$ alors X_i est une **occurrence de non-terminal** sinon X_i est une **occurrence de terminal**.

Nous associons à chaque occurrence de non-terminal X_i le type syntaxique $\mathcal{X}_i$ et à chaque occurrence de terminal X_j, le type sémantique $\mathcal{X}_j$. Nous noterons de façon générique $type(X_i) = \mathcal{X}_i$.

Par abus de langage nous confondons une production avec son nom ou avec sa production nommée associée.

La notion de grammaire étant maintenant précisée, nous pouvons introduire les attributs, mais avant il nous faut donner quelques notions importantes.

Définition 2 (Signature) *Une signature $\Sigma = (\mathcal{K}, \mathcal{F})$ est composée de $\mathcal{K}$ un ensemble de types et de $\mathcal{F}$ un ensemble de fonctions. $\mathcal{F}$ est muni de deux applications :*

- $\alpha\ : \mathcal{F} \rightarrow \mathcal{K}^*$, *$\alpha(f)$ désigne le profil de la fonction f ;*
- $\tau\ : \mathcal{F} \rightarrow \mathcal{K}$, *$\tau(f)$ désigne le type de la fonction f ;*

Définition 3 (Termes et Abstractions) *Soit $\mathcal{V}$ un ensemble de variables typées sur $\mathcal{K}$; on appelle terme sur Σ de type T :*

- *une variable de type T ;*
- *la donnée d'une fonction $f \in \mathcal{F}$ avec $\alpha(f) = T$ et $\tau(f) = \alpha_1 \cdots \alpha_n$, $n \geq 0$ et la donnée d'un ensemble de termes $t_1 \cdots t_n$ de types respectifs $\alpha_1 \cdots \alpha_n$. On note ce terme $f(t_1, \cdots, t_n)$.*

On appelle abstraction d'un terme t, l'opérateur abstrait (fonction) $F_t(x_1, \cdots, x_n)$ tel que les x_i sont des variables distinctes qui apparaissent au moins une fois dans t.

Nous appelons fonctions imbriquées un terme construit à partir d'au moins deux fonctions.

Nous pouvons maintenant décrire les grammaires attribuées.

Définition 4 (Grammaire Attribuée) *Une grammaire attribuée est composée des éléments suivants :*

- *une CF-grammaire,* $G = (N, T, P, \mathcal{Z})$ *;*
- *une signature* $\Sigma = (\mathcal{K}, \mathcal{F})$*, avec* $T \subset \mathcal{K}$ *;*
- *pour tout non-terminal* $\mathcal{X} \in N$*, un ensemble d'***attributs** $\mathcal{A}(\mathcal{X})$*, union disjointe de* $\mathcal{S}(\mathcal{X})$*, l'ensemble des attributs* **synthétisés** *de* $\mathcal{X}$ *et de* $\mathcal{H}(\mathcal{X})$*, l'ensemble des attributs* **hérités** *de* $\mathcal{X}$*.*

  Chaque attribut a est de type : type(a) $\in \mathcal{K}$*. On impose* $\mathcal{S}(\mathcal{Z}) = \{z\}$ *et* $\mathcal{H}(\mathcal{Z}) = \emptyset$*. Par extension, à chaque* **occurrence de non-terminal** X_i *dans une production de la forme* $p : X_0 \to X_1 \cdots X_n$*, on associe l'ensemble d'***occurrences d'attributs** *suivant :* $A(X_i) = S(X_i) \cup H(X_i)$ *avec* $X_i.a \in A(X_i)$ *si* $a \in \mathcal{A}(\mathcal{X}_i) = \mathcal{A}(type(X_i))$*. Chaque occurrence d'attribut est une variable.*

- *pour tout terminal* $\mathcal{W} \in T$*, les ensembles d'attributs* $\mathcal{A}(\mathcal{W}) = \mathcal{S}(\mathcal{W}) = \{\epsilon\}$ *et* $\mathcal{H}(\mathcal{W}) = \emptyset$ *avec* ϵ *de type sémantique* $\mathcal{W}$*.*

  Par extension, à chaque occurrence de terminal X_i *dans une production de la forme* $p : X_0 \to X_1 \cdots X_n$*, on associe les ensembles d'occurrences d'attributs suivants :* $A(X_i) = S(X_i) = \{X_i.\epsilon\}$ *que l'on notera aussi* $\{X_i\}$[1] *et* $H(X_i) = \emptyset$ *avec* $\{\epsilon\} = \mathcal{A}(\mathcal{X}_i)$*. La valeur de l'unique attribut du non-terminal est la valeur sémantique du terminal si elle existe.*

On définit :

$$DO(p) = S(X_0) \cup \left(\bigcup_{1 \leq i \leq n} H(X_i) \right)$$

l'ensemble des occurrences d'attributs définies dans p ;

$$UO(p) = H(X_0) \cup \left(\bigcup_{1 \leq i \leq n} S(X_i) \right)$$

l'ensemble des occurrences d'attributs utilisées dans p ;

$$W(p) = UO(p) \cup DO(p)$$

l'ensemble des occurrences d'attributs de p.

[1] Nous emploierons plusieurs fois la notation $.\epsilon$ avec la propriété que $a.\epsilon = a$

- *pour toute production $p : X_0 \to X_1 \cdots X_n$, un ensemble E_p de règles sémantiques. Pour tout $X_k.a \in DO(p)$ il existe une unique règle sémantique dans E_p définissant cette occurrence d'attribut, notée :*

$$r_{p,a,k} : X_k.a = F_{p,a,k}(X_{f(1)}.a_1, \cdots, X_{f(q)}.a_q)$$

avec $X_{f(i)}.a_i \in UO(p)$, f une fonction de $\{1, \cdots, q\}$ dans $\{0, \cdots, n\}$ et $F_{p,a,k}$ l'abstraction d'un terme pour l'ensemble de variables $\mathcal{V}$ composé de l'ensemble des occurrences d'attributs utilisées dans p.

Nous appelons *classe* d'un attribut le fait qu'il soit hérité ou synthétisé. Nous parlerons également de *classe opposée*; la classe opposée d'hérité (resp. de synthétisé) est synthétisé (resp. hérité).

Nous avons restreint notre définition aux grammaires attribuées en forme normale, c'est-à-dire dont les occurrences d'attributs en partie droite des règles sémantiques sont toutes dans $UO(p)$. Cette contrainte ne diminue pas le pouvoir d'expression puisqu'il est possible, à partir d'une grammaire attribuée qui n'est pas en forme normale, d'obtenir une grammaire attribuée équivalente en forme normale [10].

En attachant un seul attribut à la racine nous ne diminuons pas non plus le pouvoir d'expression, cet attribut pouvant être constitué d'un ensemble de valeurs.

Comme nous en aurons besoin plus loin, nous introduisons dès maintenant la notion d'attribut local.

Définition 5 (Attribut local) *On appelle attribut local a attaché à une production p, une variable typée qui est définie à partir d'attributs d'entrée sur la production p par une unique règle sémantique. Cet attribut ne peut apparaître que dans les règles sémantiques associées à p.*

Comme nous désirons que, comme les attributs normaux, les attributs locaux soit toujours définis, mais également qu'ils permettent une certaine souplesse de définition, ceux-ci sont attachés aux productions. En effet, s'ils étaient attachés, comme c'est parfois le cas, à un non-terminal et qu'ils ne soient accessibles que sur les productions dont ce non-terminal est le père, il faudrait soit permettre qu'ils ne soient pas définis sur certaines productions, soit leur donner une définition "bidon" sur ces mêmes productions.

Cette différence est minime et n'induit pas de grande différence théorique.

Arbre d'Entrée

Un arbre d'entrée t pour un grammaire attribuée est un arbre syntaxique associé à une phrase du langage $L(G)$.

On note u un nœud de l'arbre t et l un nœud feuille de celui-ci. A chaque nœud u on associe une production $label(u) = p : X_0 \to X_1 \cdots X_n$ telle que :

- u est de type syntaxique (est associé au non-terminal) $\mathcal{X}_0 = type(X_0)$;

- u a n fils, on dit que u est d'arité n ;
- si u_i est le nœud fils de u en position i alors u_i est de type syntaxique $\mathcal{X}_i$;
- si l_j est le nœud feuille fils de u en position j alors l_j est de type sémantique (est associé au terminal) $\mathcal{X}_j$;
- la fonction $Pos(u, j)$ renvoie le nœud fils de u en position j.

On dit que :

$$u \rightarrow \cdots l_j \cdots u_i \cdots$$

est une instance de la production $label(u)$.

Nous parlerons de nœud ou de nœud feuille en position i pour le fils de u en position i et de nœud en position 0 pour u.

De façon générique, le type syntaxique d'un nœud u est noté $\mathcal{U} \in N$ et le type sémantique d'un nœud feuille l, $\mathcal{L} \in T$.

Évaluation

L'arbre d'entrée est la donnée d'une grammaire attribuée. L'évaluation des attributs sur un arbre d'entrée t, consiste à évaluer l'ensemble des attributs associés à chaque nœud de l'arbre d'entrée conformément à leurs types et aux règles sémantiques de la grammaire attribuée.

Plus précisément :

- à chaque nœud u de l'arbre d'entrée t on associe un ensemble d'**instances d'attributs** :
$$IA(u) = \{u.a | a \in \mathcal{A}(\mathcal{U})\} ;$$
- à chaque nœud feuille l de t, on associe le singleton :
$$IA(l) = \{l.\epsilon\}$$
l'occurrence d'attribut $l.\epsilon$ a la valeur du nœud feuille ;
- à chaque nœud u et à chaque règle sémantique de $E_{label(u)}$ de la forme :
$$r_{label(u),a,k} : X_k.a = F_{label(u),a,k}(X_{f(1)}.a_1, \cdots, X_{f(q)}.a_q)$$
on associe l'équation :
$$u_k.a = F_{label(u),a,k}(x_{f(1)}.a_1, \cdots, x_{f(q)}.a_q)$$
où u_k représente le nœud en position k et x_i représente de façon générique le nœud ou le nœud feuille en position i.

Nous définissons pour un arbre t l'ensemble de ses instances d'attributs $IA(t)$ comme l'ensemble contenant toutes les instances d'attributs associées à un nœud ou à un nœud feuille de t.

Nous obtenons finalement un système d'équations $K(t)$ dont les inconnues sont les instances d'attributs de l'arbre d'entrée t.

Évaluer les attributs sur t, c'est résoudre le système $K(t)$.

Nous nous restreignons en fait au calcul du résultat de la grammaire attribuée pour un arbre t, aussi appelé *valeur sémantique*, qui est l'unique instance d'attribut synthétisé z de la racine. Le problème revient donc à calculer uniquement les instances d'attributs qui sont nécessaires au calcul de z.

Nous appelons impasse statique tout attribut qui quelque soit l'arbre d'entrée ne sert jamais au calcul du résultat de la grammaire attribuée.

Nous appelons impasse dynamique toute instance d'un attribut qui n'est pas une impasse statique et telle qu'il existe un arbre dans lequel elle ne sert pas au calcul du résultat.

Le système $K(t)$ a une unique solution s'il n'est pas circulaire. On dit qu'une grammaire attribuée est non-circulaire ou bien formée, si quelque soit l'arbre d'entrée t, le système $K(t)$ n'est pas circulaire.

Dans ce cas, il est possible moyennant un réordonancement des équations de construire un système "triangulaire supérieur". Il existe alors un ordre simple d'évaluation des instances d'attributs.

Cette méthode de résolution semble la plus simple et la plus générale, malheureusement elle n'est pas très viable étant donné la place et le temps qu'elle nécessite. C'est pourquoi différents auteurs se sont penchés sur différentes méthodes d'évaluation des attributs.

2.2 Les méthodes d'évaluation

Nous allons commencer par introduire la notion de graphe de dépendance qui est utile pour la description des différents types évaluateurs.

Définition 6 (Graphe de Dépendance Locale) *On associe à chaque production de la forme $p : X_0 \to X_1 \cdots X_n$ un graphe de dépendance locale $D(p)$ où l'ensemble des sommets est l'ensemble des éléments de $W(p)$ et tel qu'il existe un arc allant de $X_j.b$ vers $X_i.a$ si et seulement s'il existe une règle sémantique de la forme :*

$$r_{p,a,i} : X_i.a = F_{p,a,i}(\cdots X_j.b \cdots).$$

Par abus de langage, nous confondons ce graphe de dépendance locale avec la relation qui lui est associé $\xrightarrow{p}$[2].

Par extension on définit le graphe de dépendance globale sur un arbre t comme l'union des instanciations des graphes de dépendance locale sur l'ensemble des instances de productions de t. On note $\xrightarrow{t}$ sa relation associée.

[2]Étant donnée une relation R (ou son graphe), R^+ dénote la fermeture transitive de R.

Une première famille d'évaluateurs est basée sur la méthode dite méthode dynamique, très proche de la résolution du système. Elle consiste à trouver dynamiquement pour un arbre d'entrée t l'ordre sur les instances d'attributs induit par $\xrightarrow{t}$ et à calculer les instances d'attributs en allant des plus petites aux plus grandes au sens $\xrightarrow{t}^{+}$. Cette méthode très simple a été utilisée dans de nombreux systèmes [11–18]. La classe des grammaires attribués acceptées par cette méthode est la classe des grammaires attribuées non-circulaires. Il est possible de tester statiquement la non-circularité d'une grammaire attribuée. Malheureusement les évaluateurs basés sur cette méthode sont très peu efficaces car pour chaque arbre d'entrée ils doivent reconstruire l'ordre. D'autre part le test statique de non-circularité d'une grammaire attribuée est exponentiel en la taille de la grammaire [19].

Par opposition à cette méthode de nombreux auteurs ont cherché à déterminer de façon "statique" l'ordre d'évaluation.

Une première approche utilisée dans la famille des évaluateurs dits "par passes" consiste à fixer d'une façon arbitraire la stratégie de parcours de l'arbre et l'ordre de calcul des instances d'attributs. Un certain nombre de systèmes sont basés sur cette approche [1]. Les évaluateurs de cette famille sont très efficaces, mais la classe des grammaires attribuées acceptées est très restreinte.

Une troisième famille d'évaluateurs basée sur des méthodes dites statiques essayent à partir des graphes de dépendance locale de déduire statiquement un ordre d'évaluation des occurrences d'attributs d'une production valable quelque soit l'arbre d'entrée. Les différentes méthodes se différencient par le type de l'ordre qu'elles construisent. Par opposition à la famille précédente où l'ordre était fixé de façon arbitraire, ici c'est la grammaire attribuée par sa structure qui donne l'ordre. La classe des grammaires attribuées acceptées est donc plus large.

Nous allons nous intéresser plus particulièrement dans ce mémoire à cette dernière famille d'évaluateurs. Ce choix se justifie car le produit efficacité des évaluateurs, classe des grammaires attribuées qu'ils acceptent, semble être le meilleur pour cette famille d'évaluateurs. D'autre part ce travail se place au sein du système FNC-2 [20] qui produit des évaluateurs basés sur une méthode statique.

Nous allons maintenant décrire de façon précise les classes de grammaires attribuées associées aux différentes méthodes statiques. Nous donnerons également une idée des évaluateurs attachés à chacune d'elles.

La classe des grammaires attribuées FNC

La classe des grammaires attribuées fortement non-circulaires (FNC) semble être, d'après les travaux de M. Jourdan [21–23], la classe la plus large permettant l'utilisation d'une méthode statique. Cette méthode statique est dérivée de la construction théorique de B. Courcelle et P. Franchi-Zannettacci [24,25]. Elle est basée sur le résultat suivant :

Résultat 1 *Soit une grammaire attribuée et un arbre d'entrée pour celle-ci. La valeur d'une instance d'attribut synthétisé attachée à un nœud donné de l'arbre ne dépend que :*

- *des valeurs des instances d'attributs hérités de ce nœud ;*
- *du sous-arbre issu de ce nœud.*

La validité de ce résultat est une conséquence évidente de la notion de calcul local à une instance de production. Il est naturel d'associer à chaque attribut synthétisé $a \in \mathcal{A}(\mathcal{X})$, une fonction d'évaluation prenant comme paramètres un arbre syntaxique t ayant pour racine r avec $type(r) = \mathcal{X}$ et un ensemble de valeur d'attributs hérités de $\mathcal{X}$. Cette fonction renvoie la valeur de l'instance d'attribut $r.a$ attachée à la racine de t, étant données les valeurs de certaines instances d'attributs héritées associées à r (cf. figure 2.1).

Pour calculer le résultat de la grammaire attribuée pour un arbre d'entrée t, il suffit alors d'appeler la fonction d'évaluation associée à z en lui passant t comme paramètre. En effet l'ensemble des attributs hérités de z est vide. Il reste à déterminer l'ensemble des attributs hérités à passer en paramètres de chacune des fonctions d'évaluation. En général, il n'est pas possible de passer l'ensemble des attributs hérités du non-terminal $\mathcal{X}$ car ceci risque d'entraîner des circularités dans les calculs.

L'ensemble des attributs hérités nécessaires au calcul d'un attribut synthétisé d'un même non-terminal dépend du sous-arbre issu de ce non-terminal. C'est une notion dynamique. Le sélecteur d'arguments qui est une approximation des relations réelles entre les attributs hérités et synthétisés d'un non-terminal permet de déterminer statiquement si on peut construire pour chaque attribut synthétisé un ensemble d'attributs hérités suffisant pour calculer la valeur de cet attribut synthétisé et n'induisant pas de circularité dans les calculs.

La classe des grammaires attribuées sur lesquelles il est possible de construire cet ensemble à partir du sélecteur d'arguments, est appelée classe des grammaires attribuées FNC.

Plus précisément, le sélecteur d'arguments définit une famille de relations d'ordre partiel : $\mathcal{R}_{FNC}(\mathcal{X})_{\mathcal{X} \in N}$ sur les différents $\mathcal{A}(\mathcal{X})$ telles que $\mathcal{R}_{FNC}(\mathcal{X}) \subseteq \mathcal{H}(\mathcal{X}) \times \mathcal{S}(\mathcal{X})$.

Étant donné une production $p : X_0 \rightarrow X_1 \cdots X_n$ et une relation $\mathcal{R}_{FNC}(\mathcal{X})$ sur $\mathcal{A}(\mathcal{X})$, on définit de façon générique les relations dérivées suivantes :

- $R_{FNC}(X_i) = \{(X_i.a, X_i.b) | (a, b) \in \mathcal{R}_{FNC}(\mathcal{X}_i)\}$ sur $A(X_i)$;
- $R_{FNC}(p) = R_{FNC}(X_0) \cup \cdots \cup R_{FNC}(X_n) \cup D(p)$ sur $W(p)$;
- $\mathcal{R}_{FNC}(p, \mathcal{X}_0) = \left[D(p) \cup \left(\bigcup_{1 \leq i \leq n} R_{FNC}(X_i) \right) \right]^+ / \mathcal{X}_0$,

  la projection sur $\mathcal{A}(\mathcal{X}_0)$ de la restriction de $\left[D(p) \cup \left(\bigcup_{1 \leq i \leq n} R_{FNC}(X_i) \right) \right]^+$ aux éléments de $A(X_0)$.

Il est alors possible de définir de façon précise la classe des grammaires attribuées FNC.

Définition 7 (Classe FNC) *Une grammaire est FNC si et seulement s'il existe une famille de relations d'ordre partiel* $\mathcal{R}_{FNC}(\mathcal{X})_{\mathcal{X} \in N}$ *telle que pour toute production de la forme* $p : X_0 \to X_1 \cdots X_n$ *on a les propriétés suivantes :*

- **clôture** *:* $\mathcal{R}_{FNC}(p, \mathcal{X}_0) \subseteq \mathcal{R}_{FNC}(\mathcal{X}_0)$ *;*
- **non-circularité** *:* $R_{FNC}(p)$ *sans circuit.*

A chaque relation R on attache un graphe associé dont les sommets sont les éléments de l'ensemble sur lequel R porte et tel qu'il existe un arc de a vers b si et seulement si $(a, b) \in R$. Les arcs vont donc dans le sens du transport d'informations et non dans le sens "dépend de". Par abus de langage nous confondons la relation et le graphe.

A chaque attribut synthétisé a sur un non-terminal $\mathcal{X}$ on associe :

$$\mathcal{R}_{FNC}(\mathcal{X}, a) = \{b \in \mathcal{H}(\mathcal{X}) | (b, a) \in \mathcal{R}_{FNC}(\mathcal{X})\}.$$

La propriété de clôture implique que pour tout attribut synthétisé a sur $\mathcal{X}$, $\mathcal{R}_{FNC}(\mathcal{X}, a)$ contient tous les attributs hérités h tels qu'il existe un arbre où a dépend de h. En d'autres termes, les dépendances induites du sélecteur d'arguments doivent contenir toutes les dépendances réelles. La propriété de non-circularité permet de vérifier qu'il n'existe pas de dépendance induite qui crée une circularité dans le calcul. En conséquence de quoi, il existe certaines grammaires attribuées (cf. figure 2.2) pour lesquelles il est impossible de trouver un sélecteur d'arguments qui vérifie les propriétés FNC, bien que celles-ci soient non-circulaires. Cette perte en pouvoir d'expression est très faible puisque toutes les grammaires attribuées "pratiques" que nous connaissons par expérience sont FNC.

Il existe un algorithme polynomial qui calcule pour une grammaire attribuée FNC une famille de relations d'ordre partiel $\mathcal{R}_{FNC}(\mathcal{X})$. Cette algorithme est à la fois un test d'appartenance à la classe FNC et un algorithme de construction des relations d'ordre partiel. Nous l'appelons algorithme FNC (cf. figure 2.3).

Les $\mathcal{R}_{FNC}(\mathcal{X})$ construits sont minimaux au sens de l'inclusion. Il existe *a priori* pour chaque $\mathcal{X}$ un ensemble de relations $\mathcal{R}_{FNC}(\mathcal{X})$ valides. Pour des raisons d'efficacité il est préférable de prendre la plus petite.

Lorsqu'une grammaire attribuée est FNC il est possible de construire un évaluateur FNC composé d'un ensemble de fonctions mutuellement récursives [23].

L'ordre d'évaluation trouvé statiquement est *a priori* un ordre partiel. L'ordre effectif d'évaluation est déterminé dynamiquement sur un arbre d'entrée donné par la structure même des règles sémantiques. Du fait du sélecteur d'arguments qui n'est qu'une approximation des relations réelles, la méthode FNC force le calcul sur un arbre donné de certaines instances d'attributs qui ne participent pas au calcul du résultat.

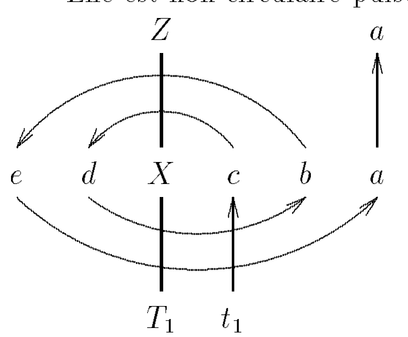
```
Fonction VISIT-a-𝒳 (nœud,h_1,···,h_m)
/* Le type syntaxique de nœud doit être 𝒳, a ∈ 𝒮(𝒳) et ℛ_FNC(𝒳,a) = {h_1,···,h_m} */
   Dans le cas où label(nœud) est
      ⋮
   p : X_0 → X_1 ··· X_n alors
      On retourne F_{p,a,0} (
            /* Avec comme paramètres */
               ⋮
            h_i,
            /* Pour les occurrences d'attribut de la forme h_i(X_0) ∈ H(X_0) */
               ⋮
            fils(nœud,k);
            /* Pour les occurrences d'attribut telles que X_k
                                    est une occurrence de terminal */
               ⋮
            VISIT-s_j-𝒳_{f(j)} (fils(nœud,f(j)),
                  F_{p,z_1,f(j)} (···),
                     ⋮
                  F_{p,z_{m'},f(j)} (···))
            /* Pour les autres occurrences d'attribut de la forme
                  s_j(X_{f(j)}) ∈ S(X_{f(j)}) et avec ℛ_FNC(𝒳_{f(j)},s_j) = {z_1,···,z_{m'}} */
                     );
      ⋮
   Fin dans le cas où
Fin fonction
```

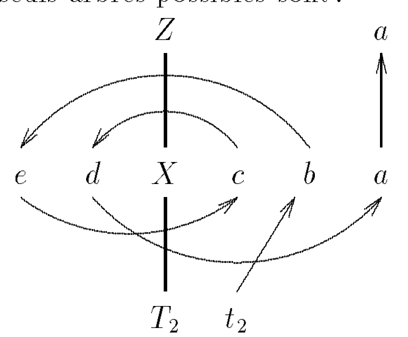
Figure 2.1: Structure d'une fonction d'évaluation pour un attribut synthétisé $a \in \mathcal{S}(\mathcal{X})$

Soit la grammaire attribuée :

$p_0\ : Z \rightarrow X$	$p_1\ : X \rightarrow T_1$	$p_2\ : X \rightarrow T_2$
$a(Z) = a(X)$;	$a(X) = e(X)$;	$a(X) = d(X)$;
$e(X) = b(X)$;	$b(X) = d(X)$;	$b(X) = t_2(T_2)$;
$d(X) = c(X)$;	$c(X) = t_1(T_1)$;	$c(X) = e(X)$;

Elle est non-circulaire puisque les deux seuls arbres possibles sont :





Pourtant elle n'est pas FNC puisque nécessairement :

$e \in \mathcal{R}_{FNC}(\mathcal{X}, c)$ et $d \in \mathcal{R}_{FNC}(\mathcal{X}, b)$

ce qui implique la circularité de $R_{FNC}(p_0)$.

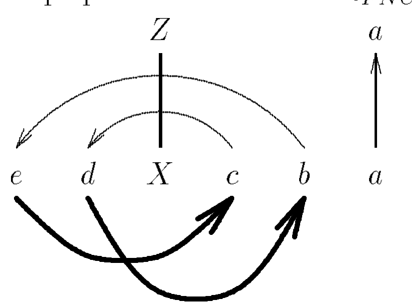


Figure 2.2: Grammaire attribuée NC non FNC

/* Calcul des $\mathcal{R}_{FNC}(\mathcal{X})$ minimaux */

Pour chaque $\mathcal{X} \in N$ **faire**
    $\mathcal{R}_{FNC}(\mathcal{X}) \leftarrow \emptyset$;
Répéter
    `convergence` $\leftarrow$ *vrai* ;
    Pour chaque $\mathcal{X}_0 \in N$ **faire**
        Pour chaque $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
$$\mathcal{R}_{FNC}(p, \mathcal{X}_0) \leftarrow \left[D(p) \cup \left(\bigcup_{1 \leq i \leq n} R_{FNC}(X_i) \right) \right]^+ / \mathcal{X}_0 \,;$$
            $\mathcal{R} \leftarrow \mathcal{R}_{FNC}(p, \mathcal{X}_0)$;
            Si $\mathcal{R} \not\subseteq \mathcal{R}_{FNC}(\mathcal{X}_0)$ **alors**
                $\mathcal{R}_{FNC}(\mathcal{X}_0) \leftarrow \mathcal{R}_{FNC}(\mathcal{X}_0) \cup \mathcal{R}$;
                `convergence` $\leftarrow$ *faux* ;
            Fin si
        Fin pour chaque
    Fin pour chaque
Jusqu'à `convergence`;

/* Test FNC proprement dit */

Pour chaque $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
    Si `circulaire(`$R_{FNC}(p)$`)` **alors**
        Stop "La grammaire n'est pas FNC" ;
    Fin si
Fin pour chaque
Écrire "La grammaire est FNC" ;

Figure 2.3: Algorithme FNC

Il y alors deux possibilités pour l'évaluation. Soit aucune instance d'attribut n'est stockée dans l'arbre, celles-ci sont vues comme les valeurs de retour de fonctions et donc certaines valeurs d'occurrences d'attributs sont recalculées, soit elles sont stockées sur les noeuds de l'arbre et il est nécessaire de tester avant d'utiliser leur valeur si elles ont déjà été calculées. Le problème de la première solution est que chaque utilisation d'une instance d'attribut nécessite le recalcul complet de sa valeur, ce qui fait perdre en efficacité. La deuxième évite cet inconvénient mais nécessite une grande place mémoire.

La classe des grammaires attribuées DNC

La méthode d'évaluation sur la classe DNC [24] est comparable dans les grandes lignes à la méthode d'évaluation sur la classe FNC. Ces deux méthodes diffèrent principalement dans le fait que l'évaluation DNC peut débuter sur n'importe quel nœud de l'arbre d'entrée. Ceci implique une évaluation ascendante de certaines instances d'attributs hérités et donc des fonctions d'évaluation qui remontent dans l'arbre d'entrée.

Cette méthode est basée sur le résultat associé à la classe des grammaires attribuées FNC et au résultat suivant :

Résultat 2 *Soit une grammaire attribuée et un arbre d'entrée pour celle-ci. La valeur d'une instance d'attribut hérité attachée à un nœud donné de l'arbre ne dépend que :*

- *des valeurs des instances d'attributs synthétisés de ce nœud ;*
- *de l'arbre englobant de ce nœud.*

L'arbre englobant d'un nœud u est l'arbre complet auquel a été enlevé le sous-arbre issu de u.

Comme pour les attributs synthétisés dans la méthode FNC, à chaque attribut hérité de $\mathcal{X}$ est associé une fonction qui ne dépend que de l'arbre englobant et d'un ensemble de valeurs d'attributs synthétisés de $\mathcal{X}$ (cf. figure 2.4). D'autre part, la notion de sélecteur d'arguments est étendue aux attributs hérités. Il faut trouver une famille de relations d'ordre partiel qui englobe les relations réelles pour tout arbre d'entrée.

Afin que le sélecteur d'arguments n'entraîne pas de circularités dans les calculs il faut que la famille des relations $\mathcal{R}_{DNC}(\mathcal{X}) \in (\mathcal{H}(\mathcal{X}) \times \mathcal{S}(\mathcal{X})) \cup (\mathcal{S}(\mathcal{X}) \times \mathcal{H}(\mathcal{X}))$ vérifie certaines propriétés.

Les définitions des relations $R_{DNC}(X_i)$ et de $R_{DNC}(p)$ sont les mêmes que celles de $R_{FNC}(X_i)$ et de $R_{FNC}(p)$ où FNC a été remplacé par DNC.

Étant donnée une production $p : X_0 \rightarrow X_1 \cdots X_n$ et les occurrences de non-terminal X_i de type syntaxique $\mathcal{X}_i$, on définit de façon générique la relation :

$$\mathcal{R}_{DNC}(p, \mathcal{X}_i) = \left[D(p) \cup \left(\bigcup_{\substack{0 \leq j \leq n \\ j \neq i}} R_{DNC}(X_j) \right) \right]^+ / \mathcal{X}_i$$

qui est la projection sur $\mathcal{A}(\mathcal{X}_i)$ de la restriction de $\left[D(p) \cup \left(\bigcup_{\substack{0 \leq j \leq n \\ j \neq i}} R_{DNC}(X_j)\right)\right]^+$

aux éléments de $A(X_i)$.

Définition 8 (DNC) *Une grammaire attribuée est DNC si et seulement s'il existe une famille de relations d'ordre partiel $\mathcal{R}_{DNC}(\mathcal{X})_{\mathcal{X} \in N}$ telle que pour toute production $p : X_0 \rightarrow X_1 \cdots X_n$ on ait les propriétés suivantes :*

- **double clôture** *: $\forall i, 0 \leq i \leq n, \mathcal{R}_{DNC}(p, \mathcal{X}_i) \subseteq \mathcal{R}_{DNC}(\mathcal{X}_i)$;*
- **non-circularité** *: $R_{DNC}(p)$ sans circuit.*

Comme pour la clôture pour la définition FNC, la double clôture implique que toutes les dépendances réelles soient dans les $\mathcal{R}_{DNC}(\mathcal{X})$.

Il existe un algorithme polynomial [3,26] qui permet de caractériser la classe DNC et de calculer la famille des $\mathcal{R}_{DNC}(\mathcal{X})$ (cf. figure 2.5). On remarque que pour calculer cette famille, la grammaire attribuée doit déjà être FNC et les $\mathcal{R}_{DNC}(\mathcal{X})$ doivent contenir les sélecteurs d'arguments FNC.

Cette classe semble être la classe la plus large pour laquelle il existe un ordre partiel connu statiquement permettant une évaluation ascendante. Cette classe est, d'après les remarques précédentes, une sous-classe de la classe FNC. Il existe des grammaires attribuées qui sont FNC sans être DNC (cf. figure 2.6).

Pour toute grammaire attribuée DNC on est capable de construire un évaluateur DNC qui est capable de remonter dans l'arbre [27,28]. Il a une forme qui est très proche de celle de l'évaluateur FNC. Comme pour les évaluateurs FNC, l'ordre global d'évaluation n'est connu que dynamiquement pour un arbre d'entrée donné.

La classe des grammaires attribuées l-ordonnée

Contrairement aux deux méthodes statiques présentées précédemment la méthode d'évaluation l-ordonnée est basée sur le fait que l'on connaît statiquement l'ordre total d'évaluation des attributs sur une production.

Cette classe de grammaires attribuées a été introduite par P. Deransart [29], mais aussi par J. Engelfriet [30] sous le nom simple multi-visites.

Définition 9 (l-ordonnée) *Une grammaire attribuée est l-ordonnée si et seulement s'il existe une famille de relations d'ordre total $\mathcal{R}_l(\mathcal{X})_{\mathcal{X} \in N}$ telle que pour chaque production p, $R_l(p)$ soit sans circuit.*

Fonction `VISIT-`a`-`$\mathcal{X}$ `(nœud,`$\mathtt{s}_1$`,`$\cdots$`,`$\mathtt{s}_m$`)`
`/*` Le type syntaxique de `nœud` doit être $\mathcal{X}$, $a \in \mathcal{H}(\mathcal{X})$ et $\mathcal{R}_{DNC}(\mathcal{X}, a) = \{s_1, \cdots, s_m\}$ `*/`
  Dans le cas où *label*`(père(nœud))` **est**
    $\vdots$
  $p : X_0 \rightarrow X_1 \cdots X_n$ **alors**
    `pos` $\leftarrow$ `position(nœud);`
    On retourne $F_{p,a,\mathtt{pos}}$ `(`
        `/*` Avec comme paramètres `*/`
          $\vdots$
        $\mathtt{s}_i$`,`
        `/*` Pour les occurrences d'attribut de la forme $s_i(X_{\mathtt{pos}}) \in S(X_{\mathtt{pos}})$ `*/`
          $\vdots$
        `fils(père(nœud),`k`)`
        `/*` Pour les occurrences d'attribut telles que X_k
                est une occurrence de terminal `*/`
          $\vdots$
        `VISIT-`s_j`-`$\mathcal{X}_{f(j)}$ `(fils(nœud,`$f(j)$`)),`
            $F_{p,z_1,f(j)}$ `(`$\cdots$`),`
              $\vdots$
            $F_{p,z_{m'},f(j)}$ `(`$\cdots$`))`
        `/*` Pour les autres occurrences d'attribut de la forme
              $s_j(X_{f(j)}) \in S(X_{f(j)})$ et avec $\mathcal{R}_{DNC}(\mathcal{X}_{f(j)}, s_j) = \{z_1, \cdots, z_{m'}\}$ `*/`
        `/*` On a nécessairement $f(j) \neq$ `pos */`
          $\vdots$
        `VISIT-`h_j`-`$\mathcal{X}_0$ `(père(nœud),`
            $F_{p,z_1,f(j)}$ `(`$\cdots$`),`
              $\vdots$
            $F_{p,z_{m'},0}$ `(`$\cdots$`))`
        `/*` Pour les occurrences d'attribut de la forme
              $h_j(X_0) \in H(X_0)$ et avec $\mathcal{R}_{DNC}(\mathcal{X}_0, h_j) = \{z_1, \cdots, z_{m'}\}$ `*/`
              `);`
    $\vdots$
  Fin dans le cas où
Fin fonction

Figure 2.4: Structure d'une fonction d'évaluation pour un attribut hérité $a \in \mathcal{H}(\mathcal{X})$

/* Calcul des $\mathcal{R}_{DNC}(\mathcal{X})$ minimaux */

/* On apelle l'algorithme FNC et on génère ainsi la famille de relations d'ordre partiel $\mathcal{R}_{FNC}(\mathcal{X})$ */
Pour chaque $\mathcal{X} \in N$ **faire**
  $\mathcal{R}_{DNC}(\mathcal{X}) \leftarrow \mathcal{R}_{FNC}(\mathcal{X})$;
Fin pour chaque
Répéter
  `convergence` $\leftarrow$ *vrai* ;
  Pour chaque $\mathcal{X}_0 \in N$ **faire**
    Pour chaque $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
      Pour *pos* **de** 1 **à** n **faire**

$$\mathcal{R}_{DNC}(p, \mathcal{X}_{pos}) \leftarrow \left[D(p) \cup \left(\bigcup_{\substack{0 \leq j \leq n \\ j \neq i}} R_{DNC}(X_j) \right) \right]^+ / \mathcal{X}_i \;\; ;$$

        $\mathcal{R} \leftarrow \mathcal{R}_{DNC}(p, \mathcal{X}_{pos})$;
        Si $\mathcal{R} \nsubseteq \mathcal{R}_{DNC}(\mathcal{X}_{pos})$ **alors**
          $\mathcal{R}_{DNC}(\mathcal{X}_{pos}) \leftarrow \mathcal{R}_{DNC}(\mathcal{X}_{pos}) \cup \mathcal{R}$;
          `convergence` $\leftarrow$ *faux* ;
        Fin si
      Fin pour
    Fin pour chaque
  Fin pour chaque
Jusqu'à `convergence`;

/* Test DNC proprement dit */

Pour chaque $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
  Si `circulaire(`$R_{DNC}(p)$`)` **alors**
    Stop "La grammaire n'est pas DNC" ;
  Fin si
Fin pour chaque ;
Écrire "La grammaire est DNC" ;

Figure 2.5: Algorithme DNC

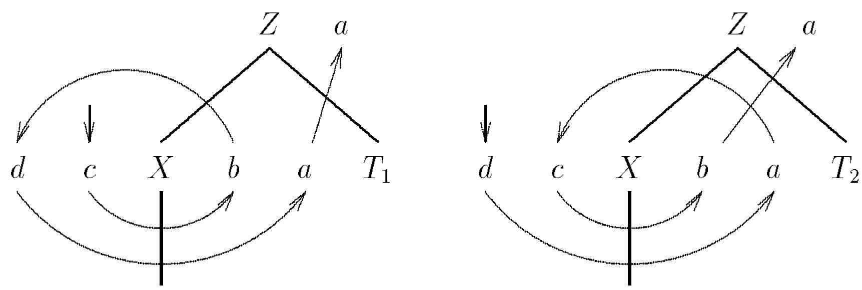
Soit la grammaire attribuée :

$p_0 : Z \rightarrow X\, T_1$	$p_1 : Z \rightarrow X\, T_2$	$p_2 : X \rightarrow$
$a(Z) = a(X);$	$a(Z) = b(X);$	$a(X) = d(X);$
$d(X) = b(X);$	$c(X) = a(X);$	$b(X) = c(X);$
$c(X) = f();$	$d(X) = g();$	

Cette grammaire attribuée est FNC puisque la seule relation non vide est :

$$\mathcal{R}_{FNC}(\mathcal{X}) = \{(d, a), (c, b)\}$$

et celle-ci contient exactement les dépendances réelles de p_2 et donc n'induit pas de circularité dans les calculs car la grammaire attribuée est NC. En effet les deux seuls arbres possibles sont :



Pourtant elle n'est pas DNC puisque nécessairement :

$a \in \mathcal{R}_{DNC}(\mathcal{X}, c)$ et $b \in \mathcal{R}_{DNC}(\mathcal{X}, d)$

ce qui implique la circularité de $R_{DNC}(p_2)$.

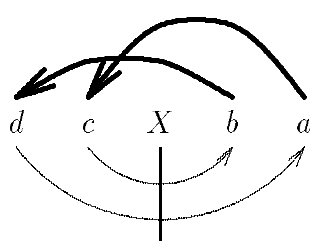


Figure 2.6: Grammaire attribuée FNC non DNC

Si une grammaire attribuée est de classe l-ordonnée, la relation $\xrightarrow{t}^{+}$ est une relation d'ordre total sur les instances d'attributs de l'arbre t. Il est possible de coder l'évaluateur sous forme de séquences de calculs, appelées séquences de visites dans [30,31] respectant les ordres $R(p)$ (cf. figure 2.7). Celles-ci nous permettent de construire des évaluateurs totalement déterministes pour les grammaires attribuées l-ordonnées. Contrairement aux méthodes précédentes il n'est pas nécessaire de tester si une instance d'attribut a déjà une valeur avant de la calculer.

V-Z-1(u)
    E-X_1-b(*fils*$(u, 1)$)
    V-X-1(*fils*$(u, 1)$)
    E-X_1-a(*fils*$(u, 1)$)
    V-X-2(*fils*$(u, 1)$)
    E-Y_2-a(*fils*$(u, 2)$)
    V-Y-1(*fils*$(u, 2)$)
    E-Z_0-c(u)
END-V

V-Z-2(u)
    E-Z_0-du)
END-V

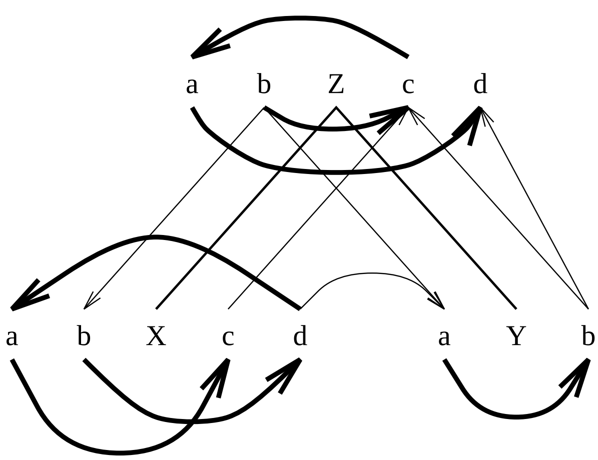


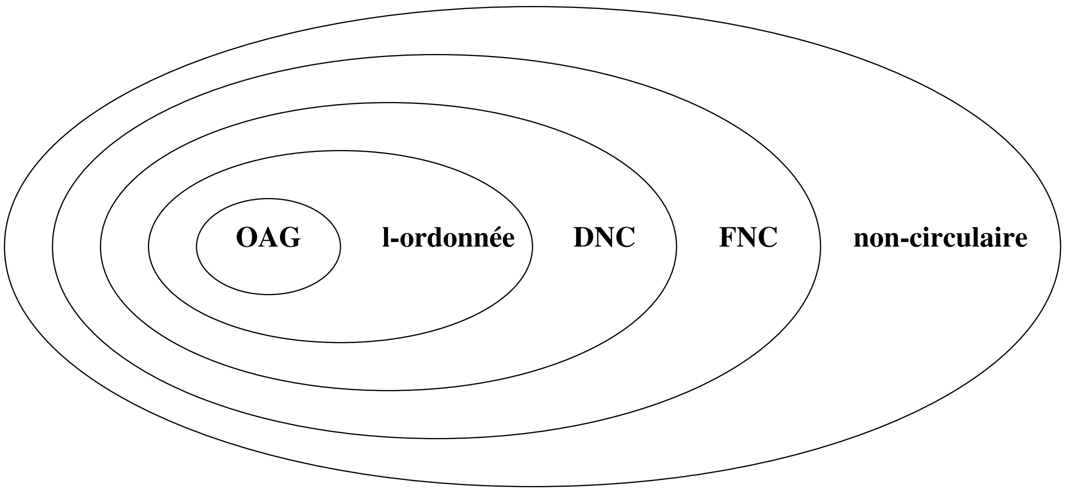
Figure 2.7: Exemple de séquence de visite

La classe des grammaires attribuées l-ordonnées est incluse dans la classe des grammaires attribuées DNC. En effet l'ordre total induit la condition de double-clôture [29,30] de la définition DNC.

Le test d'appartenance à la classe l-ordonnée ainsi que la construction des ordres totaux sont malheureusement des problèmes NP-complets [32]. Il est donc pas possible en théorie de créer un système de génération d'évaluateurs pour cette classe.

Une solution pour utiliser les propriétés intéressantes de cette classe a été de trouver une sous-classe des grammaires attribuées l-ordonnée caractérisable en un temps polynomial, la classe OAG [31].

L'ordre d'inclusion des classes de grammaires attribuées est donné par le schéma suivant :



Une autre approche qui a été développée dans le système FNC-2 est de transformer les grammaires attribuées FNC en des grammaires attribuées l-ordonnées. En effet, il a été démontré [33] que toute grammaire attribuée NC pouvait être transformée en une grammaire l-ordonnée équivalente au risque d'augmenter la taille de la grammaire attribuée de façon exponentielle.

Une transformation optimisée [34] permet dans le cas de la transformation FNC vers l-ordonnée, de faire disparaître en pratique le facteur exponentiel.

Chapitre 3

Les grammaires attribuées fonctionnelles et leur méta-composition

3.1 Présentation et définitions

Afin d'écrire une grammaire attribuée de façon modulaire, Ganzinger et Giegerich ont introduit la notion de grammaires couplées par attributs, que je nomme grammaires attribuées fonctionnelles. Celles-ci sont basées sur une interprétation des grammaires attribuées classiques. L'idée est de ne plus voir une grammaire attribuée comme un processeur qui sur un arbre d'entrée ajoute des décorations, mais comme une véritable fonction qui prend en entrée un arbre et produit un autre arbre en sortie. On appelle ces fonctions, grammaires attribuées fonctionnelles. On peut ainsi décomposer une application en une succession de grammaires attribuées fonctionnelles s'appelant en cascade.

La méta-composition est un mécanisme qui permet à partir de cette succession de grammaires attribuées fonctionnelles de construire une unique grammaire attribuée qui a la même sémantique mais qui ne construit pas les arbres intermédiaires. Il est ainsi possible d'écrire des applications de façon très modulaire sans perdre en efficacité des évaluateurs.

3.2 Les grammaires attribuées fonctionnelles

Les grammaires attribuées fonctionnelles ne sont pas une nouvelle forme de grammaires attribuées mais une nouvelle façon d'interpréter une spécification par grammaires attribuées. Les grammaires attribuées fonctionnelles suivent donc la définition donnée précédemment des grammaires attribuées.

Nous allons décrire de façon formelle une grammaire attribuée fonctionnelle α qui fait passer d'un arbre de la grammaire $G_1 = (N_1, T_1, P_1, \mathcal{Z}_1)$ vers un arbre de la grammaire $G_2 = (N_2, T_2, P_2, \mathcal{Z}_2)$, on la note $\alpha : G_1 \rightarrow G_2$.

- La syntaxe de base est G_1.

- La signature est $\Sigma = (\mathcal{K}, \mathcal{F})$.
- L'ensemble des types $\mathcal{K}$ se décompose en :
  - une partie sémantique, $\mathcal{K}_T$ ensemble de types sémantiques, avec $Bool, T_1, T_2 \subset \mathcal{K}_T$;
  - une partie syntaxique, $\mathcal{K}_N$ ensemble de types syntaxiques, $\mathcal{K}_N = N_2$.
- L'ensemble des fonctions $\mathcal{F}$ se décompose en :
  - calculs, fonctions de $\mathcal{K}_T^*$ dans $\mathcal{K}_T$;
  - constructions, fonctions de $(\mathcal{K}_N \cup \mathcal{K}_T)^*$ dans $\mathcal{K}_N$ éléments de P_2 ;
  - règles de copie, fonctions identité de $\mathcal{K}$ dans $\mathcal{K}$;
  - conditionnelles, `If_Then_Else` de $Bool \times (\mathcal{K}_N \cup \mathcal{K}_T) \times (\mathcal{K}_N \cup \mathcal{K}_T)$ dans $(\mathcal{K}_N \cup \mathcal{K}_T)$.
- Les ensembles $\mathcal{A}(\mathcal{X})$ se décomposent en :
  - attributs sémantiques, a tels que $type(a) \in \mathcal{K}_T$;
  - attributs syntaxiques, a tels que $type(a) \in \mathcal{K}_N$.
- L'attribut de la racine z est de type $\mathcal{Z}_2$.

Comme nous parlons de fonction imbriquée pour une fonction construite à partir de plusieurs fonctions élémentaires, nous parlons de construction imbriquée pour une fonction construite à partir de plusieurs constructions.

Comme les règles sémantiques sont correctement typées et que l'attribut de la racine est de type $\mathcal{Z}_2$, une grammaire attribuée de cette forme va renvoyer un résultat qui peut être interprété comme un arbre de la grammaire G_2.

Nous allons prendre l'exemple classique [4] de la grammaire attribuée $inv : G \to G$ qui prend une liste d'entiers et qui renvoie l'inverse de cette liste.

G :

$P_1 : Z_0 \to L_1 ;$ $\quad P_2 : L_0 \to L_1\ Int_2 ;$ $\quad P_3 : L_0 \to ;$

inv :

$\mathcal{S}(\mathcal{Z}) = \{z\} ; \quad \mathcal{S}(\mathcal{L}) = \{l_1\} ; \quad \mathcal{H}(\mathcal{L}) = \{l_2\} ;$

$$\begin{array}{lll} P_1 : Z_0.z = P_1(L_1.l_1) ; & P_2 : L_0.l_1 = L_1.l_1 ; & P_3 : L_0.l_1 = L_0.l_2 ; \\ \quad L_1.l_2 = P_3() ; & \quad L_1.l_2 = P_2(L_0.l_2, Int_2) ; & \end{array}$$

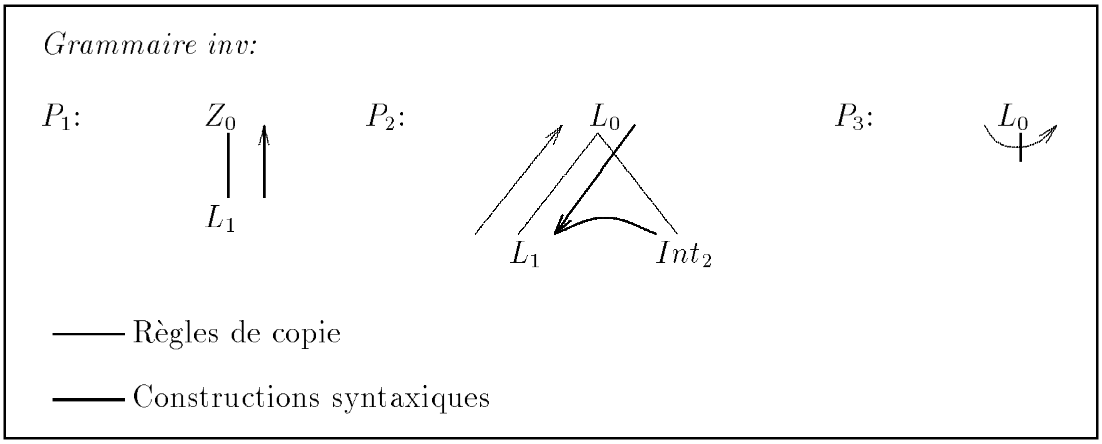


Figure 3.1: Graphe de dépendances de la grammaire attribuée *inv*

Sur un arbre donné on obtient un graphe de dépendances représenté dans la figure 3.1.

Si les constructions sont interprétées comme des constructions réelles d'arbre et les calculs comme des fonctions qui renvoient une valeur, alors z va contenir effectivement un arbre qui pourra être l'entrée d'une nouvelle grammaire attribuée fonctionnelle.

Si dans la grammaire attribuée *inv* et la fonction $P_3()$ est interprétée comme une fonction qui renvoie 0, P_2 comme une fonction qui fait la somme de ces deux arguments et P_1 comme une identité, nous obtenons une grammaire attribuée qui fait la somme des éléments de la liste. Les grammaires attribuées fonctionnelles sont donc une interprétation spéciale de grammaires attribuées classiques.

Dans le système FNC-2, la forme de la grammaire de sortie est précisée dans la spécification de la grammaire attribuée. Le système sait de façon statique si une fonction est une construction ou un calcul.

Une application peut être décomposée en une suite de grammaires attribuées fonctionnelles qui se passent des arbres les unes aux autres. Cette approche des grammaires attribuées permet une écriture modulaire d'une application. Elle peut également faciliter la spécification d'un problème par l'introduction d'une grammaire intermédiaire sur laquelle la spécification est simple.

Cette vision permet également une réutilisation de grammaire attribuée en construisant une grammaire fonctionnelle qui fait passer d'une syntaxe de base vers la syntaxe de base du problème déjà traité sous forme de grammaire attribuée (cf. chapitre 7).

La décomposition d'une application en plusieurs grammaires attribuées fonctionnelles oblige à construire des arbres intermédiaires qui ne seraient pas utiles dans une grammaire attribuée qui décrirait la totalité de l'application. L'efficacité de l'évaluateur peut être grandement diminuée par ce problème.

Ganzinger et Giegerich ont introduit [4] une construction qui permet de transformer une suite de grammaires attribuées fonctionnelles en une unique grammaire attribuée qui a la même sémantique que la suite de grammaires attribuées de départ. Nous la nommons *méta-composition.*

3.3 Composition des spécifications

L'idée de la méta-composition est simple. Prenons deux grammaires attribuées fonctionnelles $\alpha : G_1 \to G_2$ et $\beta : G_2 \to G_3$. Nous voulons éliminer la construction de l'arbre intermédiaire de G_2 ; or dans α les fonctions de construction de l'arbre de G_2 sont connues et d'autre part, grâce à β, le schéma de calcul des occurrences d'attributs associées à ces constructions est connu. Il suffit donc de remplacer, avec un renommage, les constructions de G_2 par leurs schémas de calculs associés dans β (cf. Figure 3.2). Nous obtenons ainsi une grammaire attribuée dont l'évaluateur effectue, sur les productions où auraient été construits des morceaux d'arbre, les calculs qui auraient été effectués sur ces morceaux d'arbres.

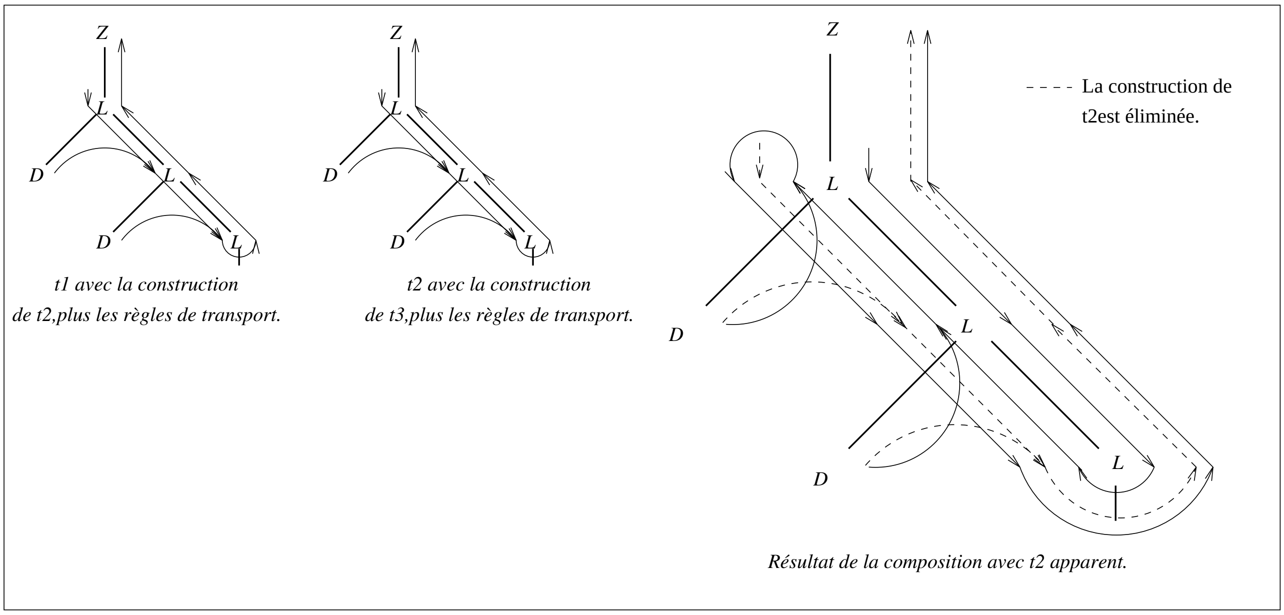


Figure 3.2: Idée de la composition

Nous allons maintenant décrire de façon précise l'algorithme de la méta-composition pour une grammaire où toutes les fonctions utilisées dans les règles sémantiques sont des fonctions élémentaires et ne comportant pas de conditionnelles (nous introduirons dans un second temps les constructions imbriquées et les conditionnelles). Ceci implique qu'à chaque abstraction de terme est associée une unique fonction de $f \in \mathcal{F}$; dans l'expression des règles sémantiques nous utiliserons le terme plutôt que l'abstraction. Il est à noter qu'une occurrence d'attribut peut alors apparaître plusieurs fois en partie droite du terme ce qui n'est pas le cas dans une abstraction.

Il faut noter que, dans cette section, ce qui nous intéresse plus particulièrement dans les règles sémantiques, c'est la structure du terme, alors que lorsque nous faisons de l'analyse statique (détermination des classes, optimisations), c'est l'abstraction qui nous intéresse.

Pour plus de simplicité nous ne différencions pas, au niveau de la notation, un terme et son abstraction.

Description de l'algorithme original

Soit donc une première grammaire attribuée fonctionnelle $\alpha : G_1 \rightarrow G_2$ et une deuxième $\beta : G_2 \rightarrow G_3$; nous allons décrire la méta-composition de ces deux grammaires attribuées, notée $\circ$. La grammaire attribuée résultante $\gamma : G_1 \rightarrow G_3$ est aussi notée $\alpha \circ \beta$.

Création de la syntaxe de base La syntaxe de base de la grammaire attribuée γ est la même que celle de α, G_1. γ doit prendre les mêmes arbres d'entrée que α. Nous gardons donc les ensembles de terminaux, de non-terminaux, de productions et l'axiome de α pour γ.

Déclaration des attributs Afin qu'il soit plus facile de différencier les ensembles d'attributs des trois grammaires attribuées, le nom de la grammaire attribuée est placé en indice. Par exemple l'ensemble des attributs synthétisés associés au non-terminal $\mathcal{X}$ dans la grammaire attribuée β est noté $\mathcal{S}_\beta(\mathcal{X})$. Cette notation sera utilisée à chaque fois que nous travaillerons avec plusieurs grammaires attribuées en même temps.

- Pour chaque attribut de type sémantique $a \in \mathcal{A}_\alpha(\mathcal{X})$, on déclare un attribut "équivalent"[1] $a \in \mathcal{A}_\gamma(\mathcal{X})$.
- Pour chaque attribut de type syntaxique $a \in \mathcal{S}_\alpha(\mathcal{X})$ et pour chaque attribut $b \in \mathcal{A}_\beta(type(a))$, on déclare un attribut de même classe et de même type que b ayant pour nom $a.b$, tel que $a.b \in \mathcal{A}_\gamma(\mathcal{X})$.
- Pour chaque attribut de type syntaxique $a \in \mathcal{H}_\alpha(\mathcal{X})$ et pour chaque attribut $b \in \mathcal{A}_\beta(type(a))$, on déclare un attribut dont la classe est l'opposée de celle de b et de même type que b ayant pour nom $a.b$, tel que $a.b \in \mathcal{A}_\gamma(\mathcal{X})$.

Création des nouvelles règles sémantiques

- Pour chaque règle sémantique de α sur une production p de la forme :

  $$r_{p,a_0,f(0)} : X_{f(0)}.a_0 = F_{p,a_0,f(0)}(X_{f(1)}.a_1, \cdots, X_{f(n)}.a_n)$$

  où $F_{p,a,k}$ est un calcul, on crée dans γ la même règle sémantique.
- Pour chaque règle sémantique de α sur une production p de la forme :

  $$r_{p,a_0,f(0)} : X_{f(0)}.a_0 = F_{p,a_0,f(0)}(X_{f(1)}.a_1, \cdots, X_{f(n)}.a_n)$$

[1] Nous dirons que deux attributs sont équivalents s'ils ont même nom, même classe et même type.

où $F_{p,a,k}$ est la construction p_2, pour chaque règle sémantique de β sur p_2 de la forme :

$$r_{p_2,b_0,g(0)} : X_{g(0)}.b_0 = F_{p_2,b_0,g(0)}(X_{g(1)}.b_1, \cdots, X_{g(q)}.b_q)$$

on construit une règle sémantique dans γ sur p de la forme :

$$r_{p,a_{g(0)}.b_0,f\circ g(0)} : X_{f\circ g(0)}.a_{g(0)}.b_0 = F_{p_2,b_0,g(0)}(X_{f\circ g(1)}.a_{g(1)}.b_1, \cdots, X_{f\circ g(q)}.a_{g(q)}.b_q)$$

avec $f \circ g(i) = f(g(i))$, fonction qui est liée à une règle sémantique.

- Pour chaque règle sémantique de α sur une production p de la forme :

$$r_{p,a_0,f(0)} : X_{f(0)}.a_0 = X_{f(1)}.a_1$$

$type(a_0) = type(a_1)$, type syntaxique.

  - Pour chaque attribut $b \in \mathcal{S}_\beta(type(a_0))$, on construit dans γ sur p une règle sémantique de la forme :

$$r_{p,a_0.b,f(0)} : X_{f(0)}.a_0.b = X_{f(1)}.a_1.b$$

  - Pour chaque attribut $b \in \mathcal{H}_\beta(type(a_0))$, on construit dans γ sur p une règle sémantique de la forme :

$$r_{p,a_1.b,f(1)} : X_{f(1)}.a_1.b = X_{f(0)}.a_0.b$$

Si nous appliquons la méta-composition à la grammaire attribuée fonctionnelle *inv* présentée dans le paragraphe précédent en la composant avec elle-même, nous obtenons la grammaire attribuée suivante :

$inv \circ inv$:
$\mathcal{S}(\mathcal{Z}) = \{z.z\}$; $\mathcal{S}(\mathcal{L}) = \{l_1.l_1, l_2.l_2\}$; $\mathcal{H}(\mathcal{L}) = \{l_1.l_2, l_2.l_1\}$

$$\begin{array}{llllll}
P_1: & Z_0.z.z = P_1(L_1.l_1.l_1)\,; & P_2: & L_0.l_1.l_1 = L_1.l_1.l_1\,; & P_3: & L_0.l_1.l_1 = L_0.l_2.l_1\,; \\
 & L_1.l_1.l_2 = P_3()\,; & & L_1.l_1.l_2 = L_0.l_1.l_2\,; & & L_0.l_2.l_2 = L_0.l_1.l_2\,; \\
 & L_1.l_2.l_1 = L_1.l_2.l_2\,; & & L_1.l_2.l_1 = L_0.l_2.l_1\,; & & \\
 & & & L_0.l_2.l_2 = P_2(L_1.l_2.l_2, INT_2)\,; & &
\end{array}$$

Le schéma de dépendances est donné par la figure 3.2.

Interprétation personnelle des résultat de l'algorithme original

Maintenant que nous avons décrit la méta-composition, il nous faut vérifier que la grammaire attribuée γ ainsi produite est bien une grammaire attribuée. Une grammaire de base, des ensembles d'attributs et des règles sémantiques ont été construits. Il suffit donc de vérifier que l'ensemble des attributs définis sur les productions ont tous une règle sémantique associée et une seule.

Si une occurrence d'attribut de type syntaxique de α n'est pas utilisée sur une production (la grammaire attribuée construit une forêt), on risque alors de créer dans γ des occurrences d'attributs non-définies (cf Figure 3.3). Si une occurrence d'attribut de type syntaxique est utilisée plusieurs fois dans α (on dit alors qu'il y a un **dag**), on risque d'avoir certaines occurrences d'attributs de γ qui sont définies plusieurs fois (cf Figure 3.4).

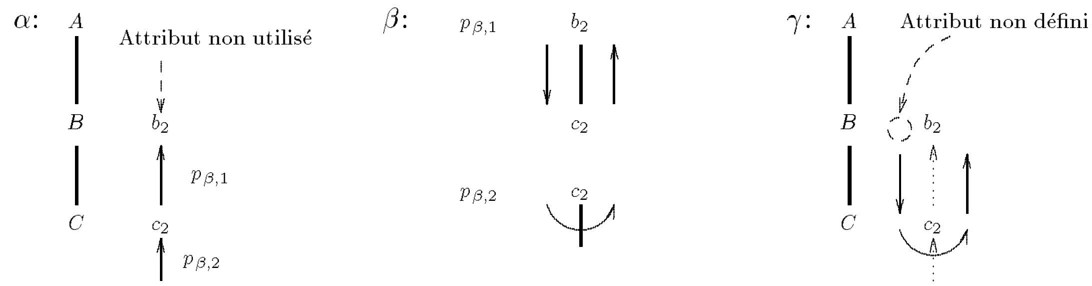


Figure 3.3: Exemple de non définition d'un attribut dans γ

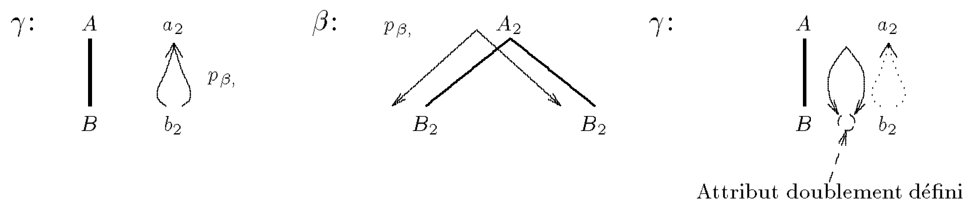


Figure 3.4: Exemple de double définition d'un attribut dans γ

Ganzinger et Giegerich ont donc restreint la classe les grammaires attribuées fonctionnelles à des grammaires attribuées qui ne contiennent pas de dags et qui ne construisent pas de forêts. Je pense que ces restrictions ne doivent pas être liées à la définition des grammaires attribuées fonctionnelles, mais plutôt qu'elles doivent définir la sous-classe des grammaires attribuées fonctionnelles acceptées en entrée de la méta-composition. En effet une grammaire attribuée avec des dags et construisant des forêts, construit tout de même un arbre comme résultat si l'attribut de la racine de l'arbre d'entrée est d'un type racine[2]. La restriction doit donc se faire au niveau de la classe d'entrée de la méta-composition et non, comme le faisaient Ganzinger et Giegerich, à l'intérieur même de la définition des grammaires attribuées fonctionnelles.

Cette vision est d'autant plus intéressante qu'elle permet de définir des conditions plus fines qui dépendent non plus uniquement de la grammaire attribuée fonctionnelle elle-même, mais également de la grammaire attribuée fonctionnelle avec laquelle elle va être composée. Nous ne cherchons plus à avoir : “quelle que soit l'utilisation que l'on en fait, on

[2]Seule condition dans la définition des grammaires attribuées fonctionnelles.

pourra appliquer sur cette grammaire attribuée fonctionnelle la méta-composition", mais : "étant donnée une utilisation particulière, peut-on appliquer la méta composition ?".

Les contraintes Nous nous plaçons toujours dans le cadre de la construction de la grammaire attribuée $\gamma = \beta \circ \alpha$.

- Il n'y a aucune contrainte directement liée à la grammaire attribuée β. Celle-ci peut tout à fait ne pas être une grammaire fonctionnelle. La seule contrainte est que la syntaxe de base de cette grammaire attribuée soit la syntaxe de sortie de la grammaire attribuée fonctionnelle α. Par habitude, elle est présentée comme une grammaire fonctionnelle, ce qui ne diminue pas le pouvoir d'expression[3].

- Il n'y a pas de contrainte sur les attributs sémantiques de α, les calculs de ces attributs n'étant pas modifiés.

- Il y a une contrainte pour les attributs de type syntaxique de α dont une des occurrences fait partie d'un dag sur une production. Soit b un tel attribut et $\mathcal{B} = \mathit{type}(b)$. Si dans β, $\mathcal{H}_\beta(\mathcal{B}) \neq \emptyset$, alors l'algorithme de méta-composition va introduire des occurrences d'attributs définies plusieurs fois. La grammaire attribuée γ construite par l'algorithme n'est pas correcte.

- Il y a une contrainte pour les attributs de type syntaxique de α dont une des occurrences n'est pas utilisée sur une production. Soit b un tel attribut et $\mathcal{B} = \mathit{type}(b)$. Si b est de classe hérité (resp. synthétisé) et si dans β, $\mathcal{H}_\beta(\mathcal{B}) \neq \emptyset$ (resp. $\mathcal{S}_\beta(\mathcal{B}) \neq \emptyset$), alors l'algorithme de méta-composition va introduire des occurrences d'attributs non définies. La grammaire attribuée γ construite par l'algorithme n'est pas correcte.

Les constructions imbriquées Les fonctions imbriquées ne posent pas de problème théorique particulier. Il suffit d'utiliser des attributs locaux correctement typés et de décomposer, grâce à ces attributs, les fonctions imbriquées en fonctions élémentaires. Les attributs locaux se comportent comme des attributs classiques pour la méta-composition. Nous verrons que ce problème est traité de façon différente dans la partie implantation.

Les conditionnelles Nous avons décidé de traiter uniquement les conditionnelles de type `if-then-else` avec une condition de type sémantique, toutes les autres conditionnelles pouvant se récrire, plus ou moins facilement, sous cette forme. Nous ne traitons pas en particulier les conditionnelles de type `case match` que l'on trouve dans le système FNC-2, dont la condition porte sur une construction d'arbre.

Dans l'algorithme de Ganzinger et Giegerich, les dags ne sont pas acceptés. Les conditionnelles de β sont donc très simples. Je pense que cette contrainte est trop forte, et même inacceptable. En effet, dans une conditionnelle une même occurrence d'attribut est souvent

[3]Il suffit de dire que l'attribut de la racine est de type racine de G_3 qui est un terminal.

utilisée dans les deux alternatives de façon différente, ce qui entraîne nécessairement un dag.

Notre position est d'accepter les dags dans les `if-then-else` de α si ces dags proviennent des deux branches du `if-then-else`. En effet, comme les deux constructions s'excluent mutuellement, il est clair qu'en gardant la condition, les ensembles de calculs associés aux deux constructions peuvent facilement être séparés. Toute double définition liée aux dags est ainsi évitée.

Nous allons donc décrire de façon précise la méta-composition sur les `if-then-else`.

- Aucun travail n'est nécessaire sur les `if-then-else` de β. En effet, la structure des règles sémantique de β n'a pas d'utilité pour la méta-composition.

- Les `if-then-else` qui apparaissent dans les règles sémantiques de α et qui ne font pas intervenir d'occurrence d'attribut de type syntaxique ne sont pas transformées. Il n'est pas non plus nécessaire de connaître la structure de ces règles sémantiques pour effectuer la méta-composition.

- Pour une règle sémantique de α où apparaissent des conditionnelles et des occurrences d'attributs de type syntaxique, les conditions sont sorties de la règle sémantique. A chacune des conditions possibles est associée une règle sémantique complète (partie droite plus partie gauche) dans laquelle aucun `if-then-else` n'apparaît. Celle-ci correspond à la règle sémantique qui aurait été obtenue dynamiquement pour une valeur de la condition. Pour cela un certain nombre de parties de la règle sémantique de départ doivent être dupliquées, notamment la partie gauche de la règle sémantique de départ.

  Par exemple la règle sémantique suivante :

  $$\begin{array}{rl} X.a := & f_1(\ \texttt{if}\ (f_{sem}(\cdots))\ \ \texttt{then}\ f_2(\cdots) \\ & \qquad\qquad\qquad\qquad\ \texttt{else}\ f_3(\cdots)) \end{array}$$

  devient après transformation :

  $$\begin{array}{rrl} \texttt{if}\ (f_{sem}(\cdots)) & \texttt{then}\ X.a := & f_1(f_2(\cdots)) \\ & \texttt{else}\ X.a = & f_1(f_3(\cdots)) \end{array}$$

Pour chacune des conditions, il existe une règle sémantique $X.a := \cdots$ appelée partie droite, elle peut être vue comme une règle sémantique classique. Il faut maintenant effectuer la méta-composition en parallèle sur ces différentes parties droites. Le problème est que certaines occurrences d'attributs n'apparaissent pas dans toutes les parties droites.

Nous commençons par appliquer la méta-composition sur chacune des parties droites r_i ; nous obtenons pour chaque r_i un ensemble de règles sémantiques $R(r_i)$. Nous construisons l'ensemble $O_\gamma(p)$ des occurrences d'attributs de γ qui apparaissent au moins une fois en partie gauche d'une règle sémantique d'un des $R(r_i)$. Pour chaque $a \in O_\gamma(p)$ et pour chaque r_i nous cherchons une règle sémantique $r'_i \in R(r_i)$ définissant a ; elle peut ne pas exister.

Nous créons alors une règle sémantique avec la partie conditionnelle à gauche et chaque r_i remplacée par r'_i si elle existe. Les `if-then-else` qui ont au moins une branche vide sont éliminés récursivement. Finalement les `if-then-else` sont replongés dans la règle pour obtenir une règle sémantique syntaxiquement correcte.

Si nous connaissons la structure des règles sémantiques de β, il est possible d'essayer de factoriser certains calculs. Si les deux branches du `if-then-else` effectuent le même calcul le `if-then-else` peut être éliminé. Dans l'implantation nous essayons de faire cette optimisation.

3.4 Optimisation du résultat

3.5 Optimisations

Lors de la méta-composition de deux grammaires attribuées fonctionnelles des suites de règles de copie peuvent apparaître. Nous donnons dans la figure 3.5 une idée graphique des suites de règles de copie introduites lors de la méta-composition avec elle-même de la grammaire *inv* présentée précédemment.

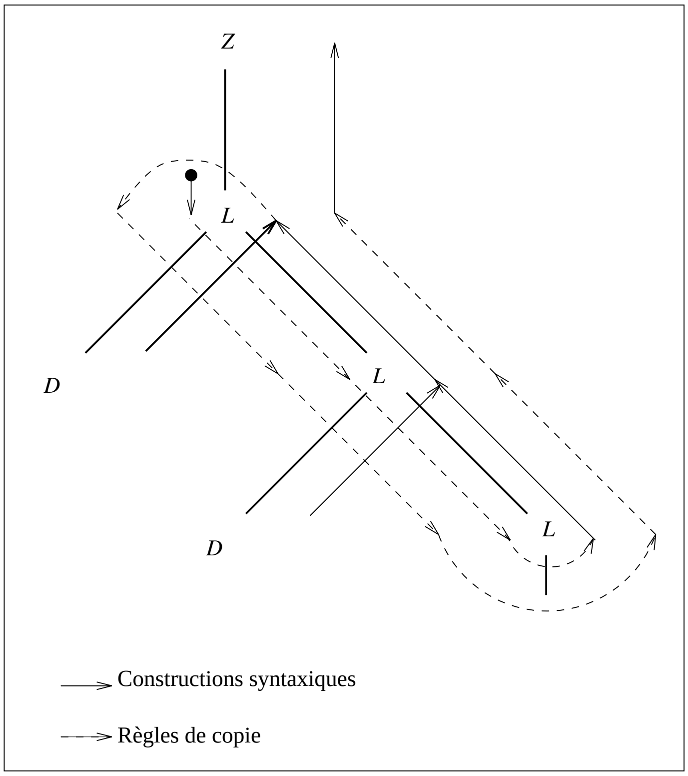


Figure 3.5: Introduction de suites de règles de copie

Notre but est de transformer la spécification de la grammaire attribuée résultante afin de faire disparaître les règles de copie inutiles.

Il est possible de les éliminer dans certains cas :

1. - *Elles commencent par une constante ;*
   - *quel que soit l'arbre d'entrée l'attribut de fin est, soit une impasse dynamique, soit égal à la constante initiale.*
2. - *Elles commencent et elles finissent dans une même production ;*
   - *quel que soit l'arbre d'entrée l'attribut de fin est, soit une impasse dynamique, soit égal à l'attribut de début.*

Nous appelons le premier cas, transport d'une constante et le second, boucle de règles de copie.

Pour éliminer les suites de règles de copie, dans les deux cas, nous commencerons par repérer les attributs susceptibles d'accepter une transformation de leur définition. Ensuite, dans chaque production, nous essaierons d'effectuer une transformation qui ne modifie pas le résultat de la grammaire attribuée, mais qui diminue la quantité de travail et, dans certains cas, le nombre de passes sur l'arbre d'entrée.

La grammaire attribuée de travail est appelée α et nous étudions comme dans la section précédente les règles sémantiques au niveau du terme et non au niveau de l'abstraction.

Pour décrire les algorithmes de repérage des attributs, il nous faut introduire la notion de *relation*. Celle-ci est très proche de la notion de dépendance du premier chapitre. Une relation est une dépendance marquée par un identificateur. L'identificateur indique si la relation a été construite en utilisant une fonction ou non.

Définition 10 (Relation) *On appelle* **relation** *un triplet* $(\perp, X_i.a, K)$ *ou* $(X_i.a, X_j.b, K)$*, tel que :* $\mathcal{X}_i, \mathcal{Y}_j \in N, a \in A(X_i), b \in A(X_j), K \in \{id, fonc\}$*. Dans une relation on notera de façon générique* $X_i.a$ *pour une occurrence d'attribut ou pour* $\perp$*.*

L'occurrence d'attribut fictive $\perp$ a été introduite pour décrire la relation à une constante. Lors de la construction de l'ensemble des dépendances, il n'est pas nécessaire de l'introduire car les constantes n'interviennent pas dans les dépendances entre attributs.

Pour chaque production nous engendrons un ensemble de relations en fonction des règles sémantiques qui se trouvent sur celle-ci.

Définition 11 (Relation engendrée) *Soit* $r_{p,a,k} \in E_p$ *une règle sémantique sur p. On définit la relation engendre notée* $\models$ *par :*

- $r_{p,a,k} \models (Y_i.b, X_k.a, id)$*, si* $r_{p,a,k} : X_k.a = Y_i.b$ *;*
- $r_{p,a,k} \models (Y_i.b, X_k.a, fonc)$*, si* $r_{p,a,k} : X_k.a = f(\cdots, Y_i.b, \cdots)$
- $r_{p,a,k} \models (\perp, X_k.a, fonc)$*, si* $r_{p,a,k} : X_k.a = f()$

Nous obtenons donc pour chaque production un graphe des relations d'une production où apparaissent toutes les relations étiquetées correctement.

Définition 12 (Graphe des relations d'une production) *On appelle* **graphe des relations d'une production** $p : X_0 \rightarrow X_1 \cdots X_n$, *l'ensemble des sommets et des arcs* $(W(p), \xrightarrow[K]{p})$.

- $W(p) = \{X_i.a | 0 \leq i \leq n, a \in A(X_i)\} \cup \{\perp\}$
- $X_i.b \xrightarrow[K]{p} X_j.a$ *ssi* $\exists r_{p,a,i} \in E_p, r_{p,a,i} \models (X_j.b, X_i.a, K)$

On le notera $D_k(p)$.

Définition 13 (Graphe des relations d'un non-terminal) *On appelle* **graphe des relations d'un non-terminal** $\mathcal{X}$, *un ensemble de sommets et d'arcs* $(V(\mathcal{X}), \xrightarrow[K]{\mathcal{X}})$

- $V(\mathcal{X}) = \mathcal{A}(\mathcal{X}) \cup \{\perp\}$
- $b \xrightarrow[K]{\mathcal{X}} a$ *avec :*
  - $(a \in \mathcal{H}(\mathcal{X})$ *et* $b \in \mathcal{S}(\mathcal{X}) \cup \{\perp\})$ *ou* $(b \in \mathcal{H}(\mathcal{X}) \cup \{\perp\}$ *et* $a \in \mathcal{S}(\mathcal{X}))$
  - $K \in \{id, fonc\}$

On le notera : $\Pi(\mathcal{X})$. *On définit le graphe des relations d'un occurrence de non-terminal* $\Pi'(X_i)$ *comme la projection du graphe des relations du non-terminal* $\mathcal{X} = type(X_i)$, $\Pi(\mathcal{X})$ *sur l'occurrence de ce non-terminal* X_i.

Ces deux notions étant précisées nous pouvons les regrouper au sein du graphe de production augmenté par I. L'ensemble I représente l'ensemble des indices des occurrences de non-terminaux de la production sur lesquelles le graphe des relations est ajouté.

Définition 14 (Graphe de production augmenté suivant I) *On appelle* **graphe de production augmenté suivant** I, *sur la production* $p : X_0 \rightarrow X_1 \cdots X_n$ *où* I *est un ensemble de numéro d'occurrence de non-terminal,* $I \subset \{0, \cdots, n\}$, *l'union du graphe des relations de* p *et des graphes des relations des* X_i, $\Pi'(X_i)$. *On le note :*

$$G_I(p) = D_k(p) \cup (\bigcup_{i \in I} \Pi'(X_i))$$

Sur ce graphe nous définissons la notion de chemin et un certain nombre de définitions qui s'y rattachent.

Définition 15 (Chemin) *On appelle* **chemin** L *dans un graphe de production augmenté, une suite de relations* $(L_i)_{1 \leq i \leq n}$, *telle que :*

$$\forall i \in \{1, \ldots, n-1\}, L_i = (X_i.a, X_j.b, K_i) \Rightarrow L_{i+1} = (X_j.b, X_k.c, K_{i+1})$$

On définit la fonction **longueur** *de* L *:* $longueur(L) = n$.
On dit que L **arrive en** $X_i.a$*, si* $L_n = (X_j.b, X_i.a, K)$.
On dit que L **passe par** $X_i.a$*, si* $\exists k \in \{1, \ldots, n\}$*,* $L_k = (X_i.a, X_j.b, K)$.
On dit que L **passe par une fonction***, si* $\exists k \in \{1, \ldots, n\}$*,* $L_k = (X_i.a, X_j.b, fonc)$.
On dit que $L \lhd G_I(p)$ *si* L *est un chemin sur* $G_I(p)$.

Nous projetons maintenant le graphe augmenté sur un non-terminal précis.

Définition 16 (Projection d'un graphe augmenté) *On appelle* **projection d'un graphe augmenté** $G_I(p)$ *sur l'occurrence d'attribut* $X_i.a$*, le graphe* $(V(\mathcal{X}), \xrightarrow[K]{\mathcal{X}})$ *où :*

- $\mathcal{X} = type(X_i)$
- $V(\mathcal{X}) = \mathcal{A}(\mathcal{X}) \cup \{\perp\}$
- $b \xrightarrow[K]{\mathcal{X}} a$ *ssi* $\exists L \lhd G_I(p)$ *avec :*
  - $L_1 = (X_i.b, X_j.c, K_1)$ *et* $L_n = (X_k.d, X_i.a, K_n)$
  - L *ne passe que par l'occurrence d'attribut de* X_i*,* $X_i.b$

  et $K = fonc$*, si* L *passe par une fonction, sinon* $K = id$.

On la note :

$$R(X_i.a) = G_I(p)^+/X_i.a$$

Si on rapporte $R(X_i.a)$ *sur le non-terminal* $\mathcal{X}_i = type(X_i)$*, on le note* $R(\mathcal{X}_i, a)$*. On appelle* **projection d'un graphe augmenté** $G_I\ (p)$ *sur* X_i *rapportée sur le non-terminal* $\mathcal{X}_i$ *:*

$$R(\mathcal{X}_i) = \bigcup_{a \in \mathcal{A}(\mathcal{X}_i)} R(\mathcal{X}_i, a)$$

On dit que $(X_i.a, X_j.b, K) \lhd D_k(p)$ ssi $\exists X_i.a \xrightarrow[K]{p} X_j.b$ dans $D_k(p)$.

Toutes ces notions étant données nous pouvons maintenant décrire les optimisations. Nous commençons par traiter les égalités à une même constante.

3.6 Élimination de règles de copie conduisant à une constante

Nous cherchons à trouver des attributs dont toutes les instances, quel que soit l'arbre d'entrée dans lequel elles se trouvent, ont toujours pour valeur une même constante. Tout d'abord nous remarquons que, si a a toujours pour valeur une même constante, ses occurrences sont définies soit par une règle de copie, soit par la constante elle-même.

Étant donnée $X_i.a$ une de ces occurrences sur une production p où $X_i.a$ ne reçoit pas directement la constante mais par une règle de copie ; alors celle-ci peut être remplacée par une égalité à la constante, cf. la figure 3.6.

Pour repérer les attributs vérifiant cette propriété, nous utilisons un algorithme simple de flot de données.

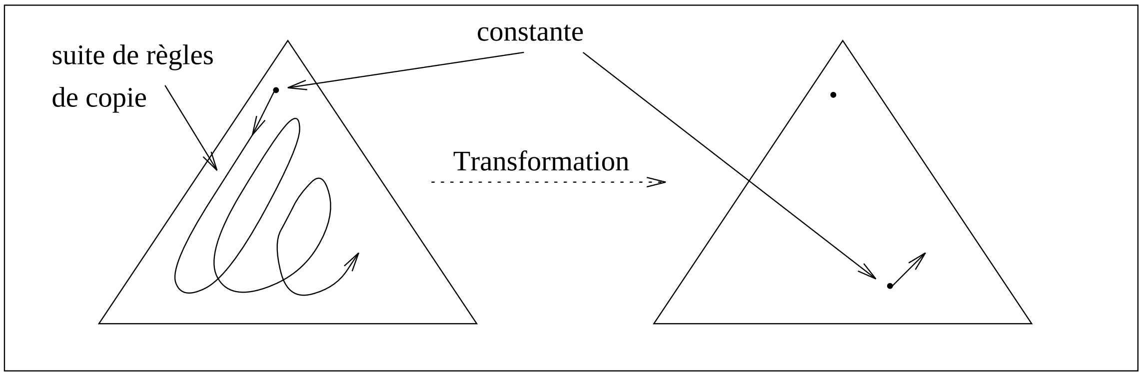


Figure 3.6: Élimination du transport d'une règle nullaire

3.6.1 L'algorithme

Le repérage

A chaque attribut a sur à $\mathcal{X}$, est associé un booléen, $\mathcal{X}.a.Bool$ et une chaîne de caractères, $\mathcal{X}.a.Nom$. A la fin de l'algorithme qui suit, si $\mathcal{X}.a.Bool$ est vrai alors a ne dépend exactement que d'une constante et $X.a.Nom$ contient le nom de cette dernière.

L'étape élémentaire de l'algorithme sur une production donnée et pour une occurrence d'attribut donnée dont le booléen est vrai, consiste à vérifier si ce booléen reste à vrai. Il ne reste à vrai que si cette occurrence est définie par une égalité à une constante ou par une règle de copie avec une occurrence d'attribut dont le booléen est à vrai et que cette nouvelle définition est compatible avec les égalités à une constante trouvées précédemment dans l'algorithme.

Si nous supposons que *Init* est un nom différent de tous les autres noms de constantes et que f est le nom de la constante qui engendre $(\perp, X_i.a, fonc)$, l'algorithme qui repère les attributs sur toute la grammaire est :

$\forall \mathcal{X} \in N, \forall a \in \mathcal{A}(\mathcal{X}), \mathcal{X}.a.Bool \leftarrow vrai, \mathcal{X}.a.Nom \leftarrow Init\,;$

Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots, X_n$ **faire**
    Pour chaque $a \in A(X_i), i \in \{0, \ldots, n\}$ **faire**
        Si $(\perp, X_i.a, fonc) \lhd D_k(p)$ **alors**
            Si $\mathcal{X}_i.a.Nom = Init$ **alors**
                $\mathcal{X}_i.a.Nom \leftarrow f\,;$
            Sinon
                Si $\mathcal{X}_i.a.Nom \neq f$ **alors**
                    $\mathcal{X}_i.a.Bool \leftarrow faux\,;$
                Fin si
            Fin si
        Fin si
    Fin pour

Fin pour

Répéter /* propagation jusqu'au point fixe */
    convergence $\leftarrow$ *vrai* ;
    Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
        Pour chaque $a \in A(X_i), i \in \{0, \ldots, n\}$ **faire**
            Si $\mathcal{X}_i.a.Bool = vrai$ **alors**
                Pour chaque $(X_j.b, X_i.a, K) \lhd D_k(p)$ **faire**
                    Si $X_j.b \neq \perp$ **alors**
                        Si $K = fonc$ **ou** $X_j.b.Bool = faux$ **alors**
                            $\mathcal{X}_i.a.Bool \leftarrow faux$;
                            convergence $\leftarrow$ *faux* ;
                        Sinon
                            Si $\mathcal{X}_i.a.Nom = Init$ **alors**
                                Si $\mathcal{X}_j.b.Nom \neq Init$ **alors**
                                    $\mathcal{X}_i.a.Nom \leftarrow \mathcal{X}_j.b.Nom$;
                                    convergence $\leftarrow$ *faux* ;
                                Fin si
                            Sinon
                                Si $\mathcal{X}_j.b.Nom \neq Init$ **et** $\mathcal{X}_j.b.Nom \neq \mathcal{X}_i.a.Nom$ **alors**
                                    $\mathcal{X}_i.a.Bool \leftarrow faux$;
                                    convergence $\leftarrow$ *faux* ;
                                Fin si
                            Fin si
                        Fin si
                    Fin si
                Fin pour
            Fin si
        Fin pour
    Fin pour
Jusqu'à *convergence*

La transformation

L'algorithme précédent nous permet de trouver des attributs dont la valeur est toujours égale à une même constante. Il est alors possible d'effectuer sur ceux-ci une transformation. Il ne reste plus, maintenant, qu'à la spécifier par l'algorithme suivant :

Pour chaque $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ **faire**
    Pour chaque $a \in A(X_i), i \in \{0, \ldots, n\}$ **faire**
        Si $\mathcal{X}_i.a.Bool = vrai$ **et** $\mathcal{X}_i.a.Nom \neq Init$ **alors**
            Si $\exists\, r_{p,a,i} : X_i.a = X_j.b \in E_p$ **alors**
                $r_{p,a,i}$ devient $r_{p,a,i} : X_i.a = \mathcal{X}_i.a.Nom()$;

Fin si
Si $\exists\, r_{p,a,i} : X_j.b = f(\ldots, X_i.a, \ldots) \in E_p$ **alors**
$r_{p,a,i}$ devient $r_{p,a,i} : X_j.b = f(\ldots, \mathcal{X}_i.a.Nom(), \ldots)$;
Fin si
Fin si
Fin pour
Fin pour

Cet algorithme de recherche nous donne un sous-ensemble des attributs pour lesquels il est possible de faire la transformation. En effet, l'algorithme que nous avons présenté donne un ensemble d'attributs dont le schéma de calcul (quel que soit l'arbre) ne passe que par des instances d'attributs, dont l'attribut associé est toujours égal à une même constante quel que soit l'arbre. En fait la condition pour pouvoir faire la transformation est beaucoup moins forte. D'autre part, il est possible d'effectuer cette transformation sur certains attributs qui ne sont pas toujours égaux à une constante. Par exemple, si a dépend, pour certains arbres, d'une même constante et pour les autres arbres d'une fonction, mais que dans ce cas, a ne participe pas au calcul du résultat de la grammaire attribuée, alors l'algorithme de transformation pourrait s'appliquer.

3.7 Élimination des boucles de règles de copie

La condition (2) de l'introduction peut très bien se réécrire :

- *il existe deux attributs, sur un même non-terminal, qui sont respectivement le début et la fin d'une suite de règles de copie.*

Si deux occurrences d'attributs d'une même production sont liées par une suite de règles de copie, non "matérialisée" dans la production, alors cette dernière sort et réentre dans la production par un même non-terminal (ceci éventuellement plusieurs fois, cf. la figure 3.7). Donc, une fois que les suites de règles de copie entre deux attributs d'un même non-terminal ont été trouvées, il est facile de retrouver les règles de copie locales à une production.

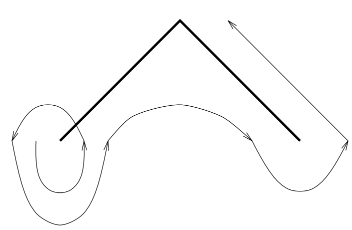


Figure 3.7: Exemple de boucle de règles de copie

Pour éliminer les suites de règles de copie arrivant sur une occurrence d'attribut $X_i.a$, cf. la figure 3.8, la propriété qu'il est nécessaire de vérifier n'est pas exactement :"$X_i.a$ ne reçoit pas une valeur dont le calcul utilise une fonction" ; mais "pour un attribut synthétisé (resp. hérité) donné du non-terminal $\mathcal{X}$, celui-ci dépend, uniquement par des règles de copie, d'un attribut hérité (resp. synthétisé) du même non-terminal, quelque soit le sous-arbre (resp. le sur-arbre)".

Il y a un rapport évident entre le traitement des attributs synthétisés (resp. hérités) et le test FNC (resp. DNC).

3.7.1 Repérage des attributs "égaux"

Algorithmes de repérage préliminaires

Avant de décrire l'algorithme complet, nous allons introduire trois algorithmes qui déterminent de façon simple si un attribut a peut être toujours égal à un autre attribut du même non-terminal. Le premier repère toujours en descendant (ou toujours en montant) si a dépend d'une fonction, le second vérifie si le schéma de calcul de a peut remonter dans l'arbre et le troisième s'il existe sur le même non-terminal que a un attribut du même type. Nous expliquerons plus précisément ces conditions dans les différents algorithmes.

A chaque attribut synthétisé (resp. hérité) a d'un non-terminal $\mathcal{X}$, on associe trois booléens. Le premier $\mathcal{X}.a.1$ prend la valeur faux si a reçoit une valeur calculée à partir d'une fonction en-dessous (resp. en-dessus). Le second $\mathcal{X}.a.2$ prend la valeur vrai si a dépend d'un hérité (resp. d'un synthétisé) par des règles de copie. Le troisième $\mathcal{X}.a.3$ est vrai s'il existe un attribut auquel a peut être égal.

Algorithme de propagation des fonctions dans un sens

S'il existe un chemin L dans l'union des $D_k(p)$ arrivant sur un attribut a, ne passant que par des relations entre synthétisés et passant par une fonction alors il existe un arbre où a dépend d'une fonction. L'attribut a ne peut donc pas être toujours égal à un autre attribut du non-terminal qui le porte. Il est facile de voir que cette conclusion peut aussi être tirée pour des attributs hérités.

1. $\forall \mathcal{X} \in N, \forall a \in \mathcal{A}(\mathcal{X}), \mathcal{X}.a.1 = \mathit{vrai}$;
   répéter
   $\mathit{convergence} \leftarrow \mathit{vrai}$;
   Pour chaque $\mathcal{X}_0 \in N$ **faire**
   Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
   Pour chaque $a \in \mathcal{S}(\mathcal{X}_0), \mathcal{X}_0.a.1 = \mathit{vrai}$ **faire**
   Si $\exists (X_j.b, X_0.a, \mathit{fonc}) \lhd D_k(p)$
   ou $\exists (X_j.b, X_0.a, \mathit{id}) \lhd D_k(p), j \in \{1, \ldots, n\}, X_j.b.1 = \mathit{faux}$ **alors**
   $X_0.a.1 \leftarrow \mathit{faux}$;
   $\mathit{convergence} \leftarrow \mathit{faux}$;
   Fin si
   Fin pour
   Fin pour

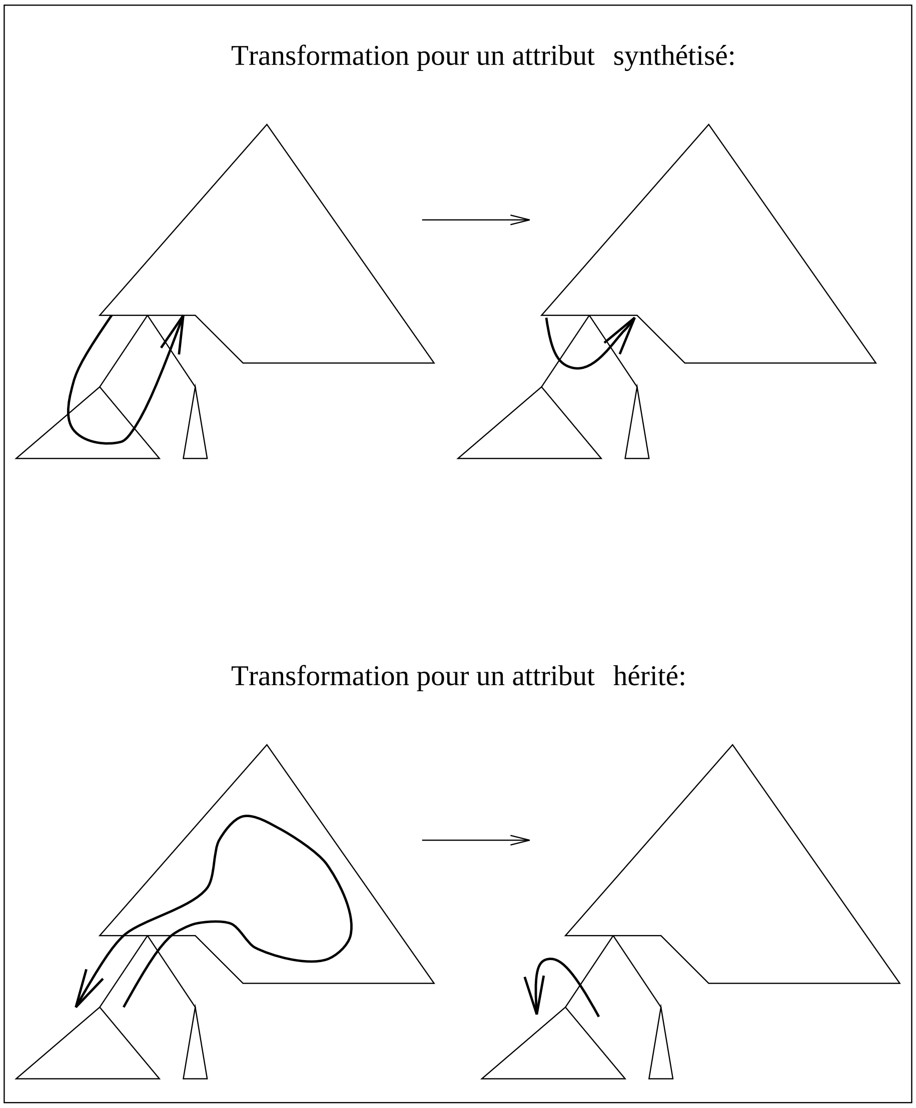


Figure 3.8: Élimination d'une suite de règles de copie

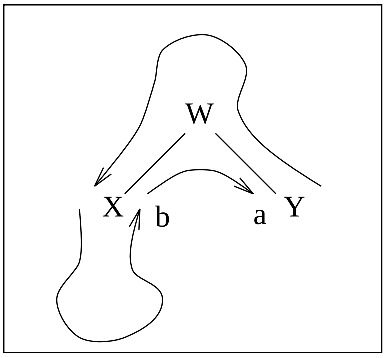


Figure 3.9: Cheminement d'une suite de règles de copie

Fin pour
Jusqu'à *convergence.*

2. **répéter**
   convergence $\leftarrow$ *vrai* ;
   Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
   Pour chaque $i \in \{1, \ldots, n\}$ **faire**
   Pour chaque $a \in \mathcal{H}(\mathcal{X}_i), \mathcal{X}_i.a.1 = \mathit{vrai}$ **faire**
   Si $\exists (X_j.b, X_i.a, \mathit{fonc}) \lhd D_k(p)$
   ou $\exists (X_j.b, X_i.a, \mathit{id}) \lhd D_k(p), j \in \{0, \ldots, n\}, j \neq i, X_j.b.1 = \mathit{faux}$ **alors**
   $\mathcal{X}_i.a.1 \leftarrow \mathit{faux}$;
   convergence $\leftarrow$ *faux* ;
   Fin si
   Fin pour
   Fin pour
   Fin pour
   Jusqu'à *convergence.*

Dans une production, un attribut hérité a d'un non-terminal $\mathcal{X}$ qui dépend d'un attribut synthétisé b d'un autre non-terminal $\mathcal{Y}$ qui a été éliminé, peut lui-même être éliminé. En effet, supposons que quelque soit l'arbre, il existe une suite de règles de copie arrivant en a et partant d'un attribut synthétisé de $\mathcal{X}$. Comme a dépend de b, il existe des cas où la suite de règles de copie passe par une instance de b. Comme l'instance de $\mathcal{Y}$ qui porte ce b n'est pas sur la même branche que l'instance de $\mathcal{X}$ qui porte a, la suite de règles de copie doit repasser, quel que soit le sous-arbre, par l'instance de $\mathcal{Y}$ (cf. la figure 3.9), ce qui implique que b est toujours égal à un autre attribut de $\mathcal{Y}$; or, si $\mathcal{Y}.b.1$ est faux, cela n'est donc pas possible.

Cet algorithme nous permet d'éliminer certaines occurrences d'attributs qui dépendent d'une fonction "par en-dessus" ou "par en-dessous". Par contre, il n'assure pas qu'un attribut synthétisé ou hérité a associé à un non-terminal X, tel que $\mathcal{X}.a.1$ est vrai, n'a pas une instance dont la valeur est calculée par une fonction.

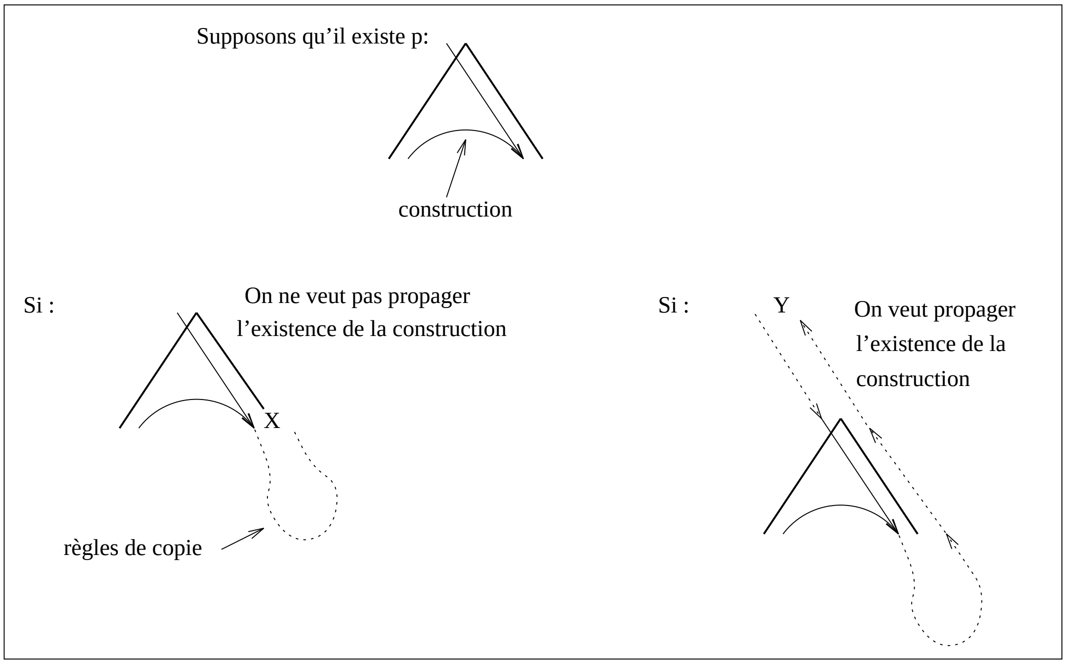


Figure 3.10: Propagation de l'information vers le bas, puis vers le haut

D'après la figure 3.10, il est clair qu'avec un algorithme de "proximité"[4], la propagation des informations vers le bas puis vers le haut (ou le contraire) n'est pas possible. En effet, avec ce type d'algorithme, dans le premier cas, si l'information qu'un attribut hérité de $\mathcal{X}$ a sa valeur calculée dans certains cas à partir d'une fonction est propagée vers le bas puis vers le haut, nous perdons l'information qu'il existe un attribut synthétisé de $\mathcal{X}$ qui reçoit sa valeur uniquement par des règles de copie. Par contre dans le deuxième cas, il va falloir faire la propagation vers le bas puis vers le haut, car une propagation uniquement vers le haut ne suffit pas. Il faut donc trouver un autre type d'algorithme.

[4]Qui ne voit qu'une production à la fois

Algorithme de propagation des cannes

Nous appelons canne, un relation du type : $(X.a, X.b, id)$. Si le calcul d'un attribut a ne passe pas par une canne, la valeur de celui-ci ne pourra pas être égale à celle d'un autre attribut du même non-terminal. En effet, la grammaire est en forme normale, donc deux attributs d'un même non-terminal ne peuvent dépendre l'un de l'autre que si l'un est hérité et l'autre synthétisé ; si cette dépendance n'est pas "matérialisée" sur un arbre par une règle sémantique dans une production, il faut bien que l'un des attributs dépende d'une telle relation "matérialisée" dans une autre production. D'autre part, cette relation synthétisé-hérité doit être une règle de copie, sinon il y a nécessairement échec (si l'on a déjà appliqué les algorithmes précédents, les cas où cette relation n'est pas une règle de copie ont déjà été éliminés). En appliquant ce critère nous obtenons l'algorithme suivant :

1. $\forall \mathcal{X} \in N, \forall a \in \mathcal{S}(\mathcal{X}), \mathcal{X}.a.2 \leftarrow faux$;
   Répéter
    $convergence \leftarrow vrai$;
    Pour chaque $\mathcal{X}_0 \in N$ **faire**
     Pour chaque $p \in P, p : X_0 \rightarrow X_1 \dots X_n$ **faire**
      Pour chaque $a \in \mathcal{S}(\mathcal{X}_0)$ avec $\mathcal{X}_0.a.2 = faux$ **faire**
       Si $\exists\,(X_j.b, X_0.a, id) \lhd D_k(p), j \in \{1, \dots, n\}, \mathcal{X}_j.b.2 = vrai$
       ou $\exists\,(X_0.b, X_0.a, id) \lhd D_k(p)$ **alors**
        $\mathcal{X}_0.a.2 \leftarrow vrai$;
        $convergence \leftarrow faux$;
       Fin si
      Fin pour
     Fin pour
    Fin pour
   Jusqu'à *convergence.*

2. $\forall \mathcal{X} \in N, \forall a \in \mathcal{H}(\mathcal{X}), \mathcal{X}.a.2 \leftarrow faux$;
   répéter
    $convergence \leftarrow vrai$;
    Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
     Pour chaque $i \in \{1, \dots, n\}$ **faire**
      Pour chaque $a \in \mathcal{H}(\mathcal{X}_i), \mathcal{X}_i.a.2 = faux$ **faire**
       Si $\exists\,(X_0.b, X_i.a, id) \lhd D_k(p), \mathcal{X}_0.b.2 = vrai$
       ou $\exists\,(X_j.b, X_i.a, id) \lhd D_k(p)$ **alors**
        Si $i = j$ **ou** $(i \neq j$ **et** $\mathcal{X}_j.b.2 = vrai)$ **alors**
         $\mathcal{X}_i.a.2 \leftarrow vrai$;
         $convergence \leftarrow faux$;
        Fin si
       Fin si
      Fin pour
     Fin pour

Fin pour
Jusqu'à *convergence.*

Algorithme de typage fort

Cet algorithme est basé sur la constatation simple qu'un attribut synthétisé (resp. hérité) ne peut recevoir la valeur d'un attribut hérité (resp. synthétisé) du même non-terminal que s'ils sont de même type. Donc pour qu'un attribut a puisse recevoir sa valeur suivant des règles de copie d'un autre attribut du même non-terminal $\mathcal{X}$ il faut qu'il existe sur $\mathcal{X}$ un attribut de même type que a mais de classe opposée. Nous proposons alors l'algorithme suivant :

$\forall \mathcal{X} \in N, \forall a \in \mathcal{A}(\mathcal{X}), \mathcal{X}.a.3 \leftarrow faux$;
Pour chaque $\mathcal{X} \in N$ **faire**
  Pour chaque $a \in \mathcal{S}(\mathcal{X})$ **faire**
    Pour chaque $b \in \mathcal{H}(\mathcal{X})$, **faire**
      Si $type(a) = type(b)$ **alors**
        $X.a.3 \leftarrow vrai$;
        $X.b.3 \leftarrow vrai$;
      Fin si
    Fin pour
  Fin pour
Fin pour

Remarques

Il est clair que ces algorithmes reflètent des conditions sur les occurrences d'attributs qui sont nécessaires mais pas suffisantes pour pouvoir effectuer des transformations.

Ces trois algorithmes peuvent très bien être appliqués séquentiellement en tenant compte des résultats déjà acquis, ce qui permet d'augmenter leur efficacité.

Algorithme de repérage des règles de copie

Nous allons maintenant décrire un algorithme qui repère des attributs qui sont toujours égaux à un autre attribut d'un même non-terminal.

L'algorithme

A chaque attribut a synthétisé (resp.hérité) sur un non-terminal $\mathcal{X}$ tel que $\mathcal{X}.a.1$, $\mathcal{X}.a.2$ et $\mathcal{X}.a.3$ soient égaux à vrai, nous cherchons à associer un graphe $Id(\mathcal{X}, a)$ tel que, quel que soit l'arc $b \xrightarrow[K]{\mathcal{X}} a$, $b \in \mathcal{A}(\mathcal{X})$ et $K = Id$. Nous notons $Id(X_i, a)$ la projection $Id(\mathcal{X}_i, a)$ sur une occurrence du non-terminal $\mathcal{X}_i$. Nous introduisons également les notations suivantes :
$Id.S(\mathcal{X}) = \bigcup_{a \in \mathcal{S}(\mathcal{X})} Id(\mathcal{X}, a)$

$Id.H(\mathcal{X}) = \bigcup_{a \in \mathcal{H}(\mathcal{X})} Id(\mathcal{X}, a)$
On note $Id.S(X_i)$ et $Id.H(X_i)$ les projections de $Id.S(\mathcal{X}_i)$ et de $Id.H(\mathcal{X}_i)$ sur l'occurrence du non-terminal $\mathcal{X}_i$.

1. **Pour chaque** $X \in N$ **faire**
    Pour chaque $a \in \mathcal{A}(\mathcal{X})$ **faire**
        Si $(\mathcal{X}.a.1$ **ou** $\mathcal{X}.a.2$ **ou** $\mathcal{X}.a.3) = \mathit{faux}$ **alors**
            $\mathcal{X}.a.1 = \mathit{faux}$;
        Fin si
    Fin pour
Fin pour

2. $\forall \mathcal{X} \in N, \forall a \in \mathcal{S}(\mathcal{X}), Id(\mathcal{X}, a) = \emptyset$;
Répéter
    $\mathit{convergence} \leftarrow \mathit{vrai}$;
    Pour chaque $\mathcal{X}_0 \in N$ **faire**
        Pour chaque $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ **faire**
            $I \leftarrow \{1, \ldots, n\}$;
            $G_I(p) \leftarrow D_k(p) \bigcup_{i \in I} Id.S(X_i)$;
            Pour chaque $a \in \mathcal{S}(\mathcal{X}_0)$ **faire**
                Si $\mathcal{X}_0.a.1 = \mathit{vrai}$ **alors**
                    Si $\exists\, L \lhd G_I(p)$ *arrivant en* $X_0.a$
                    et *(passant par* $X_i.b, i \in \{1, \ldots, n\}, b \in \mathcal{S}(\mathcal{X}_i), \mathcal{X}_i.b.1 = \mathit{faux}$
                    ou *passant par une fonction)* **alors**
                        $\mathcal{X}_0.a.1 \leftarrow \mathit{faux}$;
                    Sinon
                        $R(\mathcal{X}_0, a) \leftarrow G_I(p)^+ / X_0.a$;
                        Si $R(\mathcal{X}_0, a) \not\subseteq Id(\mathcal{X}_0, a)$ **alors**
                            $Id(\mathcal{X}_0, a) \leftarrow Id(\mathcal{X}_0, a) \bigcup R(\mathcal{X}_0, a)$;
                            $\mathit{convergence} \leftarrow \mathit{faux}$;
                        Fin si
                    Fin si
                Fin si
            Fin pour
        Fin pour
    Fin pour
Jusqu'à *convergence*

3. $\forall \mathcal{X} \in N, \forall a \in \mathcal{H}(\mathcal{X}), Id(\mathcal{X}, a) \leftarrow \emptyset$;
Répéter
    $\mathit{convergence} \leftarrow \mathit{vrai}$;
    Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
        Pour chaque $i \in \{1, \ldots, n\}$ **faire**

$J \leftarrow \{1, \ldots, n\} - \{i\}$;
$I \leftarrow J \bigcup \{0\}$;
$G_I(p) \leftarrow D_k(p) \bigcup Id.H(X_0) \bigcup (\bigcup_{j \in J} Id.S(X_j))$;
Pour chaque $a \in \mathcal{H}(\mathcal{X}_i)$ **faire**
 Si $\mathcal{X}_i.a.1 = vrai$ **alors**
  Si $\exists\, L \lhd G_I(p)\ arrivanten \mathcal{X}_i.a$
  et $(passant\ par\ X_0.b, \mathcal{X}_0.b.1 = faux, b \in \mathcal{H}(\mathcal{X}_0)$
  ou $passant\ par\ X_j.b, j \in J, b \in \mathcal{S}(\mathcal{X}_j), \mathcal{X}_j.b.1 = faux$
  ou $passant\ par\ une\ fonction)$ **alors**
   $\mathcal{X}_i.a.1 \leftarrow faux$;
  Sinon
   $R(\mathcal{X}_i, a) \leftarrow G_I(p)^+/X_i.a$;
   Si $R(\mathcal{X}_i, a) \not\subseteq Id(\mathcal{X}_i.a)$ **alors**
    $Id(\mathcal{X}_i, a) \leftarrow Id(\mathcal{X}_i, a) \bigcup R(\mathcal{X}_i, a)$;
    $convergence \leftarrow faux$;
   Fin si
  Fin si
 Fin si
Fin pour
Fin pour
Fin pour
Jusqu'à *convergence*

Terminaison :

Soit N_{nt} le nombre de non-terminaux dans la grammaire, N_a le nombre maximum d'attributs sur un non-terminal.

Le nombre maximum de dépendances que l'on peut engendrer est majoré par $N_{nt} * N_a^2$. Le nombre d'attributs est majoré par $N_{nt} * N_a$. A chaque itération avec une modification (il n'y a pas terminaison), le nombre de dépendances générées augmente ou le nombre d'attributs dont le booléen associé est à faux augmente. Comme ces deux grandeurs sont des quantités bornées le nombre d'itérations où il y a une modification est borné. Comme dès que l'on atteint un point fixe on a terminaison, c'est gagné !

On appelle arbre augmenté un arbre sur lequel on a projeté l'ensemble des graphes de relations à une itération donnée. Il est clair que les dépendances réelles sont incluses dans un tel arbre.

Démonstration :

On cherche à démontrer que s'il est possible de construire, au début d'une itération, un arbre augmenté dans lequel une instance d'attribut synthétisée, dont le booléen associé est à vrai, dépend par en-dessous d'une fonction ou d'une autre instance d'occurrence d'attribut *dont le booléen associé est à faux alors la convergence n'est pas atteinte.*

On définit la hauteur d'un chemin comme la hauteur de l'arbre minimal qui le contient. La hauteur d'un arbre étant le nombre de productions sur la plus longue de ses branches.

Soit b un attribut sur le non-terminal $\mathcal{Y}$ tel que $\mathcal{Y}.b.1 = vrai$ et a un attribut sur le non-terminal $\mathcal{X}$ tel que $\mathcal{X}.a.1 = faux$. Soit t un sous-arbre augmenté d'une instance v de $\mathcal{X}$ tel qu'il existe un chemin L reliant une instance $u.a$ de a à $v.b$.

On extrait de L le plus long sous-chemin L' arrivant en $v.b$ et ne passant que par des des instances d'attributs dont le booléen associé est à vrai. D'après la construction de L' il part sur d'une instance d'attribut $u'.a'$ dont le booléen est à faux. Il existe un chemin L'' dans sous-arbre augmenté allant de $u'.a'$ à $v.b$ ne passant que par des instances d'attributs dont le booléen est à faux et qui a une longueur minimale. Il y a alors plusieurs cas possibles :

- Soit il existe une instance d'attribut $u'.c$ synthétisée qui dépend de $u.a$ par un chemin de hauteur un. Cette dépendance directe est matérialisée dans le graphe de relation augmenté d'une instance production. Lors de "l'inspection" de ce dernier dans l'algorithme (2), le booléen associé à c passe à faux. Il n'y a pas convergence.
- Sinon, il existe un sous-chemin L''' de L'' de hauteur 2 se terminant par une instance d'attribut $u'.c$ synthétisée du père d'une instance de la production p_1 dite production "haute" et commençant par $u.a$ ou une instance d'attribut héritée du père de l'instance de p_1 (sinon le chemin L est infini, il est toujours possible de la prolonger vers le bas). On suppose que les dépendances sur les instances de productions portant L' différentes de la production "haute" sont des règles de copie entre les attributs des non-terminaux pères et qu'elles sont déjà dans leur graphe de dépendances respectif. Sinon il n'y a pas convergence.
  - Si le début du chemin L''' est $u.a$, lors de "l'inspection" du graphe augmenté de p_1, l'occurrence d'attribut de fin qui est synthétisée dépend de $u.a$. Par construction son booléen associé est à vrai, celui-ci passe alors à faux, il n'y a pas convergence.
  - Sinon, la dépendance entre l'attribut de début et celui de fin n'existait pas dans le graphe de dépendances du non-terminal père de p_1 à l'itération précédente, sinon L'' n'aurait pas été le chemin minimal (on aurait dû passer par cette dépendance plutôt que par L'). Il faut donc rajouter cette dépendance dans un graphe de dépendances. Il n'y a pas convergence.

Pour la démonstration de l'algorithme remontant l'idée est la même sauf qu'il y a un cas supplémentaire avec des chemins de hauteur trois.

3.7.2 Transformation

Il est évident que les cas pathologiques qui ne sont pas traités dans le cas de l'élimination des règles de transport d'une constante, ne peuvent pas non plus être traités ici. D'autre part, des relations induites, qui ne correspondent pas à un chemin réel, peuvent empêcher de faire une transformation alors que celle-ci serait en fait possible avec un algorithme de type NC.

Proposition 1 *S'il existe une unique relation de type Id qui arrive en $X.a$, avec $X.a.1$ égal à vrai, alors on peut effectuer une transformation sur $X.a$.*

Démonstration Supposons qu'il existe un chemin réel qui arrive en $X.a$ qui n'est pas équivalent à la relation. Si ce chemin passe par une fonction, alors $X.a.1$ vaut faux, ce qui ne correspond pas aux hypothèses. On a donc nécessairement affaire à une suite de règles de copie. Si celle-ci ne fait pas partie d'une impasse, elle repasse par un autre attribut de X. Comme le graphe de relations, par analogie avec le test FNC ou DNC, contient toutes les dépendances réelles, cela implique que l'unique relation, arrivant en $X.a$, contient le chemin précédent, ce qui est contradictoire. On peut donc conclure qu'il n'arrive en $X.a$ que des impasses ou des suites de règles de copie incluses dans la relation. On peut donc effectuer une transformation.

L'algorithme

Pour effectuer la transformation, un premier tri sur les attributs est nécessaire. A ceux qui ne dépendent que d'un autre attribut dans $Id(\mathcal{X}, a)$ on associe $\mathcal{X}.a.Test$ valant vrai, aux autres $\mathcal{X}.a.Test$ valant faux.

Pour chaque $p \in P, p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
  Répéter
    $convergence \leftarrow vrai$;
    Pour chaque $a \in \mathcal{A}(\mathcal{X}_i), i \in \{0, \ldots, n\}$ **faire**
      Si $\mathcal{X}_i.a.Test = vrai$ **alors**
        Si $\exists\, r_{p,a,i} : X_i.a = X_j.b \in E_p$
        et $j \neq i$
        et $Id(\mathcal{X}_i, a) = \{c \xrightarrow[Id]{\mathcal{X}_i} a\}$ **alors**
          $r_{p,a,i} \leftarrow r_{p,a,i} : X_i.a = X_i.c$;
          $convergence \leftarrow faux$;
        Fin si
        Si $\exists\, r_{p,b,j} : X_j.b = f(\ldots, X_i.a, \ldots), \in E_p)$
        et $Id(\mathcal{X}_i, a) = \{c \xrightarrow[Id]{\mathcal{X}_i} a\}$
        et $\exists\, r_{p,c,i} : X_i.c = g(\ldots) \in E_p$ **alors**
          $r_{p,b,j} \leftarrow r_{p,b,j} : X_j.b = f(\ldots, g(\ldots), \ldots)$;

convergence ← *faux* ;
Fin si
Fin si
Fin pour
Jusqu'à *convergence*
Fin pour.

Par cette transformation, la grammaire engendrée est bien une grammaire attribuée, car les occurrences d'attributs dont les définitions ont été modifiées sont tous bien définis. Par contre, on introduit des impasses. On pourrait appliquer l'algorithme dans l'autre sens, c'est-à-dire qu'avant de faire la transformation, on cherche les attributs dont la valeur n'est copiée que dans un seul attribut. Cela permettrait d'enlever des impasses statiques. Mais pour celles-ci, il est plus facile de propager l'information de dépendance à la racine de proche en proche et d'éliminer les attributs et les règles sémantiques qui ne contribuent pas au calcul de la valeur de la racine.

D'après la figure 3.11, on voit qu'après un première transformation, si on réapplique le test, on peut trouver de nouvelles suites de règles de copie, qui n'apparaissaient pas au premier test du fait de dépendances induites. On peut donc appliquer des transformations, tant que qu'il n'y a pas stabilité. Clairement ceci termine.

Outre le fait que cette transformation peut diminuer le nombre de passes dans l'arbre on remarque que l'on peut également changer la classe de la grammaire. Sur la figure 3.11 si l'on a une dépendances entre $X.a$ et $X.c$ avant la transformation le test FNC qui est équivalent dans ce cas au test d'égalité n'est pas vérifié. Par contre après la transformation il est vérifié.

On remarque également que suivant l'ordre dans lequel on traite les attributs, on peut obtenir des grammaires attribuées dont les spécifications sont différentes, cf. la figure 3.12. Même si l'on évalue toutes les instances d'attributs toutes les grammaires attribuées obtenues après transformation sont équivalentes au sens où toutes les instances d'attributs ont même valeur indépendamment de la grammaire attribuée.

3.7.3 Remarques générales

- La complexité de l'algorithme de recherche des boucles de règles de copie est polynomiale en la "taille" de la grammaire, comme le test FNC. Il est facile d'imaginer un algorithme de recherche basé sur celui de non-circularité. Malheureusement dans ce cas la complexité est exponentielle.

- Sur les algorithmes de repérage, on peut appliquer les notions de stabilité syntaxique et sémantique, ainsi que celle d'ordre statique [35], afin de les optimiser.

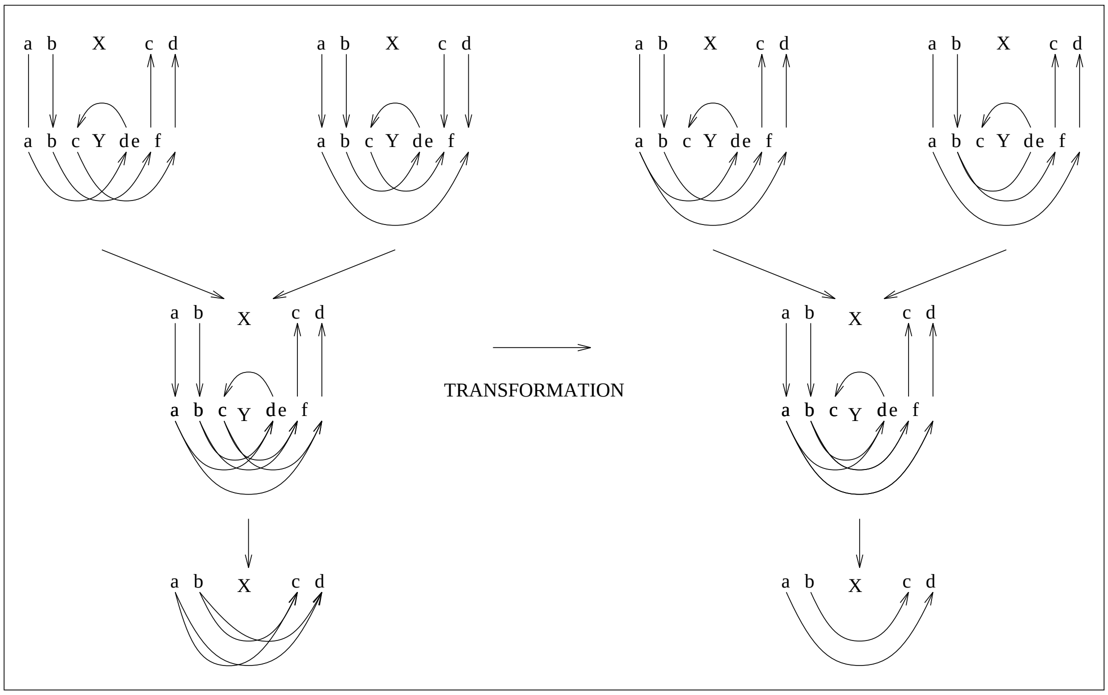


Figure 3.11: Test après transformation

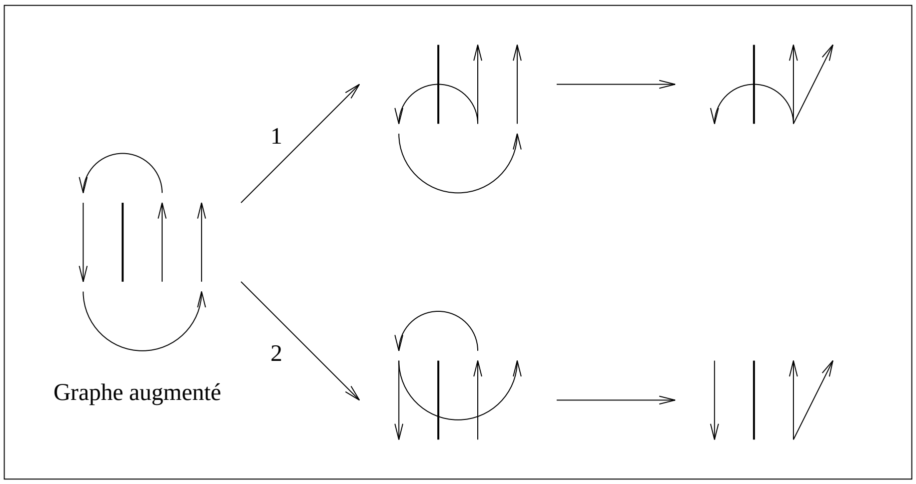


Figure 3.12: Deux transformations différentes

- On peut rapprocher le travail présenté ici de celui de Tokuda [36] ou Räihä [37], qui propose une méthode d'évaluation sur des grammaires, qui évite dynamiquement les copies, mais qui n'effectue pas à proprement parler de transformation de la grammaire.

3.8 Conclusion

Dans ce qui précède nous avons défini deux algorithmes généraux pour repérer les suites de règles de copie qui peuvent être éliminées et deux transformations pour que la grammaire attribuée soit mieux "écrite".

On peut imaginer que si on associe à la grammaire G_2 un certain nombre de règles de réécriture, il soit possible de faire le même genre de traitement, par exemple avec des fonctions nilpotentes, ou des fonctions inversibles.

Certains cas où la transformation était possible n'ont pas été traités. Avec un algorithme de type NC, on peut tous les traiter, mais avec un coût bien supérieur en temps.

Tous les algorithmes présentés ici sont implantés en sortie d'un coupleur de Grammaires Attribuées, dans le cadre du système FNC-2.

3.9 Implantation

Le système FNC-2

J'ai implanté la méta-composition au sein d'un système de traitement des grammaires attribuées appelé FNC-2 [20]. Je vais donc vous décrire brièvement ce système.

Le système FNC-2 se divise en deux parties. La première partie est purement syntaxique, elle permet de construire grâce au système SYNTAX un analyseur syntaxique pour la syntaxe d'entrée de la grammaire attribuée. Cette partie permet de construire un arbre abstrait sur lequel les calculs sémantiques vont être faits. Pour construire l'analyseur le système utilise deux spécifications : l'ASX, déclaration de la syntaxe abstraite et l'ATC (abstract tree constructor) qui permet de construire l'arbre abstrait à partir de la syntaxe concrète. La seconde partie est purement sémantique, elle comprend la déclaration des attributs et leurs définitions sur la syntaxe abstraite décrite dans l'ASX. Le système FNC-2 possède un langage OLGA dédié à cette spécification. Il va permettre de construire un évaluateur pour l'ensemble des attributs.

Lorsqu'une application est écrite au sein du système FNC-2, un analyseur syntaxique est créé grâce au système SYNTAX [38] ; il permet de construire des arbres de la grammaire abstraite décrite dans l'ASX à partir d'une phrase de la grammaire concrète décrite dans l'ATC. Il est spécifié en C ou en Le_Lisp. Ensuite, un évaluateur pour la grammaire attribuée décrite en OLGA est généré, il est de type l-ordonné si la grammaire attribuée est de classe FNC (ce qui est pratiquement toujours le cas). Cet évaluateur est également spécifié en C ou en Le_Lisp. L'analyseur syntaxique peut alors être enchaîné à un ou plusieurs évaluateurs. Le programme ainsi obtenu est capable de prendre une phrase du langage d'entrée, de construire l'arbre abstrait qui lui est associé et d'effectuer les calculs spécifiés dans la grammaire attribuée sur l'arbre ainsi construit.

Le coupleur

Le langage OLGA possède certaines particularités qui ne sont pas décrites par la méta-composition, il nous a donc fallu élargir un peu cette notion dans le coupleur. Pour cela, nous ne travaillons pas directement sur la spécification OLGA mais sur un forme interne de la grammaire attribuée. Dans cette forme interne certaines constructions particulières, non décrites dans la méta-composition, ont déjà été transformées. En particulier les conditionnelles type **case** à plusieurs entrées ont été transformées en des IF_THEN_ELSE imbriqués. Outre ces constructions classiques le système FNC-2 accepte des productions d'arité variable sur lesquelles, en plus des règles que l'on peut traiter de façon classique, il existe des conditionnelles spéciales que l'on traite au cas par cas (cf. section 3.9.1).

La forme intermédiaire est appelée AGST. Il est clair que le coupleur n'est pas capable de prendre n'importe quelle spécification OLGA, mais le sous-langage sur lequel il sait travailler à la même puissance d'expression.

D'autre part le coupleur n'a pas le même comportement que la méta-composition dans la création des règles sémantiques de la grammaire attribuée résultante. En effet, il cherche à vérifier qu'il n'y a ni dags, ni construction de forêts en même temps qu'il construit les règles sémantiques.

Je vais présenter les grandes lignes de ce mécanisme.

Nous commençons par construire l'ensemble des attributs de la grammaire attribuée résultante. Pour chaque attribut a sur $\mathcal{X}$ dans α et pour chaque attribut b sur $type(a)$ dans β on construit un attribut $a.b$ de même type que b sur $\mathcal{X}$ pour γ. Pour chacune des occurrences d'attributs de sortie d'une productions p_α nous recherchons sa définition.

- Pour les occurrences d'attributs provenant d'attributs sémantiques de α, il n'a pas de problème, on recopie la règle sémantique de α.

- Pour toute autre occurrence d'attribut $X_i.a.b$ sur une production p, elle provient d'une occurrence d'attribut de α, $X_i.a$ sur p et d'un attribut b de β déclaré sur $type(a)$.

  - A partir de $X_i.a$, on trouve une règle sémantique, r_α sur p où elle apparaît et une position dans celle-ci. S'il existe plusieurs position ou s'il r_α n'existe pas, il y a erreur (on se trouve respectivement dans la cas d'un dag ou de la construction d'une forêt dans α).

  - A partir de r_α, on trouve la production p_β qui est la construction dans r_α à laquelle $X_i.a$ participe et une position j dans cette production[5].

[5]Si l'occurrence d'attribut se trouve en partie gauche la position est 0.

- A partir de p_β et de j, on trouve la règle sémantique r_β sur p_β qui définit $Y_j.b$. Si on ne trouve pas cette règle sémantique, il y a erreur (on se trouve dans le cas où, dans α, on construit une forêt).
- La définition de $X_i.a.b$ dans γ est alors la partie droite de r_β dans laquelle les différentes occurrences d'attributs de β ont été remplacées par ceux qui leurs correspondent dans γ. Par exemple, si on cherche à remplacer $Y_k.c$.
  * Soit $X_l.d$ l'occurrence d'attribut en position k dans r_α, $Y_k.c$ devient alors $X_l.d.c$ dans γ.
  * Soit p'_β la construction qui se trouve en position k dans r_α, $Y_k.c$ devient alors l'appel à une nouvelle fonction qui est obtenu en appelant récursivement la construction que je suis en train de décrire.

On a ainsi une idée du déroulement d'une construction pour le coupleur. Je n'ai pas décrit tout les cas comme les IF_THEN_ELSE, les fonctions imbriquées comportant une partie sémantique et une partie syntaxique dans α. Ces cas sont traités dans le coupleur mais ne présentent pas d'intérêt particulier.

Il est a noté que cette construction n'introduit pas d'attributs locaux contrairement à l'algorithme de Ganzinger et Giegerich.

Le coupleur est capable de prendre en entrée une séquence de grammaire attribuée et de construire la grammaire attribuée résultante. On peut accéder à une forme simplifiée de la grammaire attribuée résultante grâce à une interface X-window, appelée Xcoupleur. Les résultats obtenus pour l'optimisation de la grammaire attribuée résultante ont aussi été implantés au sein du Xcoupleur. On peut ensuite générer une forme AGST de cette nouvelle grammaire attribuée.

Il a fallu modifier un peu le système FNC-2 afin de pouvoir générer un forme AGST adéquate et pour pouvoir, à partir de cette forme, générer un évaluateur. Le schéma du système est représenté dans la figure 3.13.

Pour l'instant le coupleur n'est qu'un prototype qui n'a été testé que sur des exemples *ad-hoc*. En effet, les grosses spécifications dont nous disposons ne sont pas des spécifications sous forme fonctionnelle, mais sous une forme que nous appelons "séquentielles" qui prend un arbre et renvoie le même arbre décoré.

Exemple d'utilisation du Xcoupleur

Nous appliquons deux fois la méta-composition de la grammaire attribuée *inv* avec elle même :

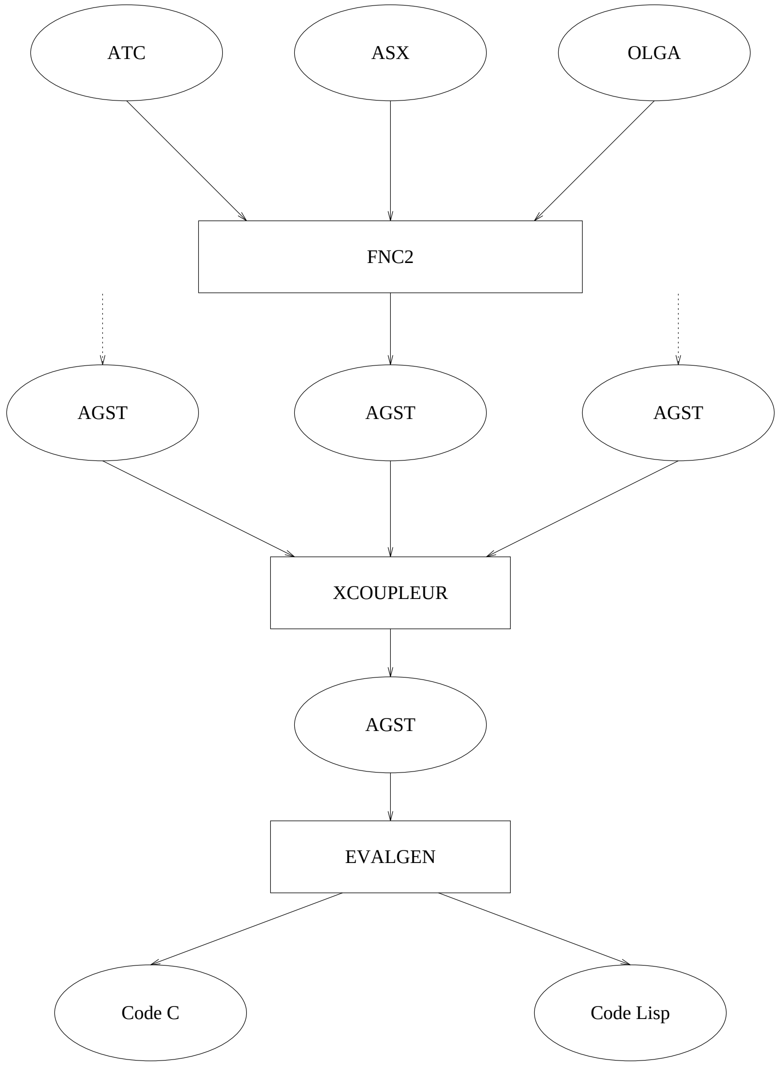
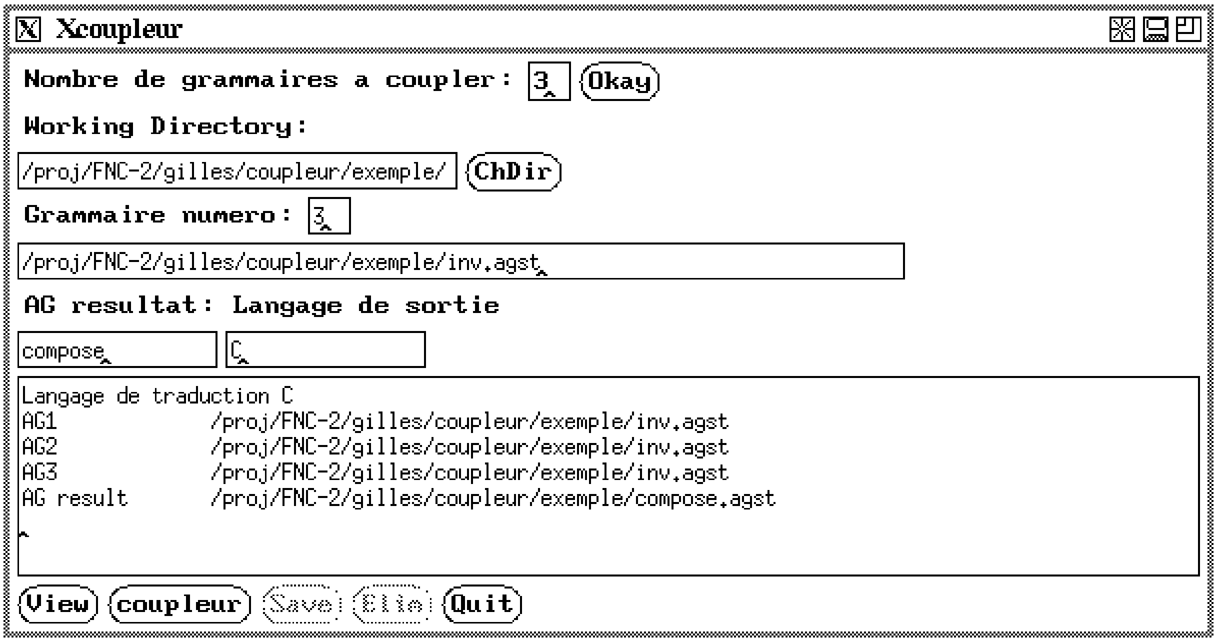
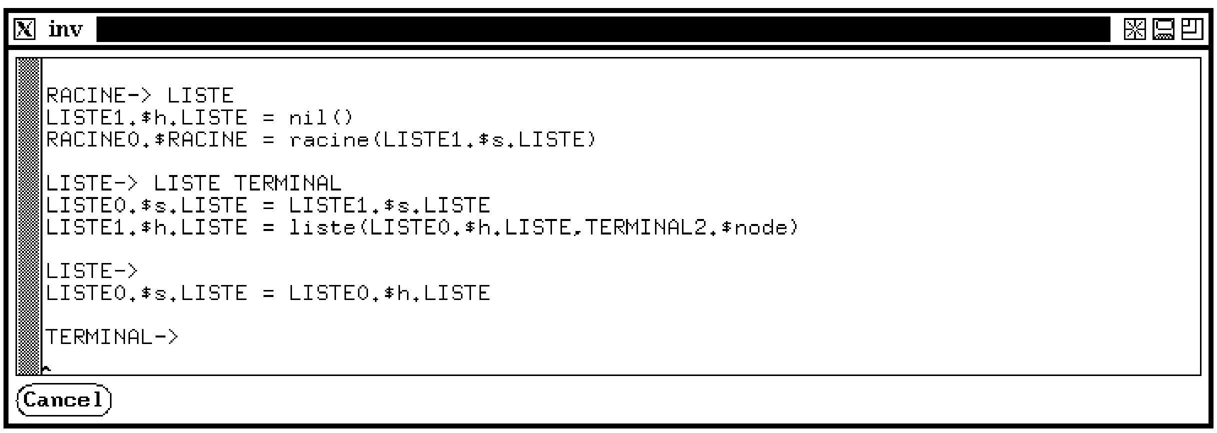


Figure 3.13: Shéma du système complet



Nous visualisons la grammaire attribuée *inv* sous la forme :



Le résultat obtenu est de la forme :

```
compose

RACINE-> LISTE
RACINE0.$RACINE.RACINE.RACINE = null-Z(LISTE1.$s.LISTE.s.LISTE.s.LISTE)
LISTE1.$h.LISTE.h.LISTE.h.LISTE = LISTE1.$h.LISTE.s.LISTE.h.LISTE
LISTE1.$h.LISTE.s.LISTE.s.LISTE = LISTE1.$h.LISTE.h.LISTE.s.LISTE
LISTE1.$s.LISTE.h.LISTE.s.LISTE = LISTE1.$s.LISTE.h.LISTE.h.LISTE
LISTE1.$s.LISTE.s.LISTE.h.LISTE = z2()

LISTE-> LISTE TERMINAL
LISTE0.$h.LISTE.h.LISTE.s.LISTE = LISTE1.$h.LISTE.h.LISTE.s.LISTE
LISTE0.$h.LISTE.s.LISTE.h.LISTE = LISTE1.$h.LISTE.s.LISTE.h.LISTE
LISTE0.$s.LISTE.h.LISTE.h.LISTE = LISTE1.$s.LISTE.h.LISTE.h.LISTE
LISTE0.$s.LISTE.s.LISTE.s.LISTE = LISTE1.$s.LISTE.s.LISTE.s.LISTE
LISTE1.$h.LISTE.h.LISTE.h.LISTE = z1(LISTE0.$h.LISTE.h.LISTE.h.LISTE,TERMINAL2.$node)
LISTE1.$h.LISTE.s.LISTE.s.LISTE = LISTE0.$h.LISTE.s.LISTE.s.LISTE
LISTE1.$s.LISTE.h.LISTE.s.LISTE = LISTE0.$s.LISTE.h.LISTE.s.LISTE
LISTE1.$s.LISTE.s.LISTE.h.LISTE = LISTE0.$s.LISTE.s.LISTE.h.LISTE

LISTE->
LISTE0.$h.LISTE.h.LISTE.s.LISTE = LISTE0.$s.LISTE.h.LISTE.s.LISTE
LISTE0.$h.LISTE.s.LISTE.h.LISTE = LISTE0.$s.LISTE.s.LISTE.h.LISTE
LISTE0.$s.LISTE.h.LISTE.h.LISTE = LISTE0.$h.LISTE.h.LISTE.h.LISTE
LISTE0.$s.LISTE.s.LISTE.s.LISTE = LISTE0.$h.LISTE.s.LISTE.s.LISTE

TERMINAL->

Cancel
```

Après optimisation celle-ci devient :

```
compose

RACINE-> LISTE
RACINE0.$RACINE.RACINE.RACINE = racine(LISTE1.$s.LISTE.h.LISTE.h.LISTE)
LISTE1.$h.LISTE.h.LISTE.h.LISTE = nil()

LISTE-> LISTE TERMINAL
LISTE0.$s.LISTE.h.LISTE.h.LISTE = LISTE1.$s.LISTE.h.LISTE.h.LISTE
LISTE1.$h.LISTE.h.LISTE.h.LISTE = liste(LISTE0.$h.LISTE.h.LISTE.h.LISTE,TERMINAL2.$node)

LISTE->
LISTE0.$s.LISTE.h.LISTE.h.LISTE = LISTE0.$h.LISTE.h.LISTE.h.LISTE

TERMINAL->

Cancel
```

3.9.1 Traitement des constructions sur les productions d'arité variable

Les constructions spéciales sur les productions d'arité variable sont :
map left, map right, case arity, case position

Le **map left** (resp.**map right**) permet d'appliquer une fonction sur un attribut des fils d'une production d'arité variable en les parcourant de gauche à droite (resp. de droite à gauche). Par exemple sur $p : A \rightarrow A*$ pour donner à l'attribut a du père de la production,

la valeur de la somme des attributs a de ces fils, on écrit :
A-father.a := **map left** +
value 0
other *A-pos.a*
end map ;

Le **case arity** permet de faire un calcul particulier pour une arité particulière de la production d'arité variable.

Le **case position** permet de faire un calcul particulier en fonction de la position du non-terminal qui porte l'attribut que l'on cherche à calculer. Il est possible de spécifier trois types de positions **first**, **last** ou **any**.

Comme nous ne savons pas traiter de façon générale ces constructions, nous allons détailler certains cas où la construction est possible. Comme précédemment nous traitons de la construction de $\gamma = \beta \circ \alpha$

Si ces constructions spéciales sont utilisées dans la partie sémantique de α, elles sont copiées sans changement.

Si dans α, on utilise les constructions d'arité variable de β avec une arité fixe, c'est-à-dire que dans les règles sémantiques les **map** ne sont pas utilisés, la méta-composition sur cette construction s'effectue de façon classique. Il suffit de faire disparaître les productions d'arité variables de β en les remplaçant par leurs instances d'arité fixe qui sont utilisées dans α.

Je restreins la méta-composition de telle sorte que les grammaires attribuées en entrée ne contiennent que des fonctions complexes non-imbriquées (un **case arity** ne peut être imbriqué avec un **map left** ...).

Nous allons détailler maintenant des cas où l'on "compose" deux fonctions complexes.

Les **map**s étant les seules constructions dans α à générer des productions listes, c'est sur celles-ci qu'il est possible de composer deux fonctions complexes.

On a le tableau de composition suivant :

$\circ$	map left	map right	case pos first	case pos last	case pos other	case arity
map left	map left	map right	case pos first	case pos last	case pos other	case arity
map right	map right	map left	case pos last	case pos first	case pos other	case arity

Je définis un opérateur $*$ sur l'ensemble {*left, right, first, last, any*}, avec la table suivante :

$*$	*left*	*right*	*first*	*last*	*any*
left	*left*	*right*	*first*	*last*	*any*
right	*right*	*left*	*last*	*first*	*any*
first	*first*	*first*	*first*	*last*	*first*
last	*last*	*last*	*first*	*last*	*last*
any	*any*	*any*	*first*	*last*	*any*

Je vais maintenant définir un certain nombre de cas de composition.

Je suppose tout d'abord que le **map** sur lequel s'effectue la composition est de la forme suivante :

$P_0 \, : A \rightarrow B^*$

A-father.a = **map** *sens* *f-arité-variable*
value *f-arité-variable()*
other *B-pos.b*
end map ;

- Composition avec un autre **map** de la forme :

  type(a)-father.a' = **map** *sens'* *f'-arité-variable*
  value *f'-arité-variable()*
  other *type(b)-pos'.b'*
  end map ;

  donne un **map** :

  A-father.a.a' = **map** *sens ∗ sens'* *f'-arité-variable*
  value *f'-arité-variable()*
  other *B-((sens ∗ pos)∗pos').b'*
  end map ;

- Composition avec un **case position** de la forme :

  type(b).b' = **case position is**
  pos'_1 : f_1*-arité-fixe(type(b)-pos''.c)*
  pos'_2 : f_2*-arité-fixe(type(a).d)*
  end case ;

  donne un **case position** :

  B.b.b' = **case position is**
  sens ∗ pos'_1 : f_1*-arité-fixe(B-(sens ∗ pos'').b.c)*
  sens ∗ pos'_2 : f_2*-arité-fixe(A-father.a.d)*
  end case ;

- Composition avec un **case arity** de la forme :

  type(a)-*father*.a' = **case arity is**
  arity: f_1-*arité-fixe*(*type*(b)-*pos*$'$.b')
  end case ;

  donne un **case arity** :

  A-father.a.a' = **case arity is**
  arity:f_1-*arité-fixe*(B-*pos*$''$.b.b')
  end case ;

  Si *pos* un numéro alors si *sens*= *left*, *pos*$''$ =*pos*$'$, sinon *pos*$''$ =*arity* $-$ *pos*$'$. Sinon *pos*$''$ =*sens* $*$ *pos*$'$.

Je suppose maintenant que le **map** est de la forme :
$P_0 : A \rightarrow B^*$

A-father.a = **map** *sens* f_1-*arité*-2
value f_2-*arité*-0()
other B-*pos*.b
end map ;

Pour que je sache faire la composition, il faut que le schéma de dépendance entre attributs soit d'une certaine forme, sur les fonctions de constructions (productions), f_1-*arité*-2 et f_2-*arité*-0. Ce sont les suivants :

- Sur f_1-*arité*-2 : $A \rightarrow AB$
  $A_0.a' = f_1'$-*arité*-2($A_1.a', B.b'$)
  Sur f_2-*arité*-0 : $A \rightarrow$
  $A_0.a' = f_2'$-*arité*-0()

  On obtient par composition avec le **map**, un nouveau **map** qui est le suivant :

  A-father.a.a' = **map** *sens* f_1'-*arité*-2
  value f_2'-*arité*-0()
  other B-*pos*.b.b'
  end map ;

- Sur f_1-*arité*-2 : $A \to AB$
  $A_1.a' = A_0.a'$
  $B.c' = f'_1$-*arité*-1$(A_1.b')$
  $A_0.b' = f'_2$-*arité*-1$(B.d')$
  Sur f_2-*arité*-0 : $A \to$
  $A.b' = f'_3$-*arité*-$(A.a')$

Ce schéma est de sens, *sens'* égal à *left*. Il a un inverse *right* que l'on sait aussi traiter. Ce dernier est obtenu en transformant les synthétisés en hérités et en inversant les relations.

On obtient par composition avec le **map**, deux règles sémantiques dont une est un **case position** :

$B.b.c'$ = **case position is**
  $((sens * sens')*first)$: $f'_1 \cdots (f'_3$-*arité*-1$(A$-*father*$.a.a'))$
  other : $f'_1 \cdots (f'_2$-*arité*-1$(B$-$((sens * sens')*left).b.d'))$
end case
A-*father*$.a.b' = f'_2$-*arité*-1$(B$-$((sens * sens')*last).b.d')$

- Sur f_1-*arité*-2 : $A \to AB$
  $A_1.a' = A_0.a'$
  $B.b' = f$-*arité*-1$(A_0.a')$
  On obtient par composition :
  $B.b.b' = f$-*arité*-1$(A$-*father*$.a.a')$

- Sur f_1-*arité*-2 : $A \to AB$
  $A_0.a' = f$-*arité*-1$(B.b')$
  On obtient par composition :
  A-*father*$.a.a' = f$-*arité*-1$(B$-$(sens * last).b.b')$

- Sur f_1-*arité*-2 : $A \to AB$
  $A_0.a' = A_1.a'$ $A_1.b' = f'_1$-*arité*-1$(B.c')$
  Sur f_2-*arité*-0 : $A \to$
  $A.a' = f'_2$-*arité*-1$(A.b')$
  On obtient par composition :
  A-*father*$.a.a' = f'_2$-*arité*-1$(f'_1$-*arité*-1$(B$-$(sens * first).b.c')$

Nous avons beaucoup restreint la classe des grammaires attribuées fonctionnelles acceptées par le coupleur. Quoi qu'il en soit, il semble que dans les cas pratiques les différents cas proposés ci-dessus soient amplement suffisants. En effet, la puissance d'expression est la même, toutes les constructions qui ne sont pas acceptées peuvent se réécrire sous une forme qui elle est acceptée moyennant une écriture quelquefois moins élégante.

3.9.2 Les attributs importés et les attributs exportés

Outre ses constructions particulières, le langage OLGA permet également l'utilisation d'attributs particuliers, dits *exporté*s et *importé*s. Les attributs importés sont des attributs qui contiennent une valeur qui a été calculée en dehors de la grammaire attribuée et qui a été laissée dans l'arbre ; les attributs exportés sont des attributs destinés à être importés dans une autre grammaire attribuée et qui donc sont laissés dans l'arbre.

Il faut donc adapter le coupleur afin d'accepter de telles constructions.

Comme précédemment nous traitons de la construction de $\gamma = \beta \circ \alpha$.

Les attributs importés et exportés de β ne posent pas de problème, les attributs de γ construits à partir de ceux-ci restent importés et exportés.

Par contre pour les attributs de α, il y a une construction particulière.

Attribut importé Si un attribut a de α est importé alors tout attribut synthétisé b sur $type(a)$ donne un attribut $a.b$ importé dans γ ; tout attribut hérité b sur $type(a)$ donne un attribut $a.b$ dans exporté dans γ.

Cette construction n'a de sens que si l'attribut importé de α est obtenu à partir d'une grammaire attribuée fonctionnelle qu'il est possible de composer avec β.

Sinon, les seuls cas où il est possible d'accepter un attribut importé est quand celui-ci est de type sémantique ou quand celui-ci ne porte pas d'attribut dans β. Il se comporte alors comme un terminal.

Attribut exporté Si un attribut a de α est exporté alors tout attribut synthétisé b sur $type(a)$ donne un attribut $a.b$ exporté dans γ ; tout attribut hérité b sur $type(a)$ donne un attribut $a.b$ importé dans γ.

Comme précédemment, cette construction n'a de sens que si l'attribut exporté est utilisé dans une grammaire attribuée fonctionnelle qu'il est possible de composer avec β.

Sinon, le seul cas où il est possible d'accepter un attribut exporté est quand celui-ci est de type sémantique.

3.9.3 Conclusion

Avec ces différentes extensions de la méta-composition appliquées dans le coupleur, on peut espérer traiter la plupart des grammaires attribuées fonctionnelles et donc pouvoir appliquer la méta-composition de façon efficace.

La seule chose qu'il est possible de regretter dans cette implantation de la méta-composition est qu'il n'est pas possible de modifier la grammaire attribuée produite. Cette contrainte est due au fait que le langage OLGA est trop complexe et que nous sommes donc passés par une forme simplifiée AGST qui n'est pas modifiable à la main. D'autre part, cet option va dans le même sens que l'idée de départ de la méta-composition qui suppose qu'il est plus simple de spécifier une application par modules que dans une "grosse" grammaire attribuée, et par conséquent que l'utilisateur ne veut pas regarder la "grosse" grammaire attribuée produite.

Chapitre 4

Composition des évaluateurs

4.1 Une première approche

4.1.1 Introduction

Étant donnée une application décomposée en une séquence de grammaires attribuées fonctionnelles, la méta-composition telle que nous venons de la présenter permet de construire automatiquement une grammaire attribuée équivalente à la séquence. Outre le fait qu'il est plus agréable de spécifier une application importante par modules, il est souvent plus facile de décrire un problème en passant par un arbre intermédiaire sur lequel le problème se spécifie de façon simple. La méta-composition, en construisant une grammaire attribuée globale pour l'application, évite la construction des arbres intermédiaires. Elle permet donc *a priori* d'avoir un évaluateur pour cette application plus efficace que l'appel successif aux différents évaluateurs des grammaires attribuées fonctionnelles de la séquence, tout en gardant les avantages d'une spécification par modules.

La méta-composition ne conserve pas, en général, les classes FNC, DNC et l-ordonnée. Donc si nous voulons être sûrs de pouvoir construire un évaluateur pour toute grammaire attribuée résultante dans le cadre FNC, il est nécessaire, dans certains cas, d'utiliser une méthode dynamique. En terme d'efficacité des évaluations, il n'est pas clair qu'il soit plus intéressant d'utiliser l'évaluateur basé sur une méthode dynamique de la grammaire attribuée résultante plutôt que l'appel successif aux évaluateurs[1] des différentes des grammaires couplées par attributs de la séquence.

Je propose dans cette section une méthode qui, dans le cadre FNC, permet toujours de construire un évaluateur basé sur des séquences de visites pour la grammaire attribuée résultante. Cette construction conserve l'avantage de ne pas construire les arbres intermédiaires tout en assurant une évaluation efficace.

Cette méthode est une approche tout à fait nouvelle, bien qu'ayant des liens très fort avec la méta-composition.

[1]Ces évaluateurs sont basés sur une méthode statique comme les grammaires couplées par attributs sont FNC.

Je généralise ensuite cette construction afin d'arriver à une véritable compilation séparée des différentes grammaires attribuées fonctionnelles qui composent une séquence.

4.1.2 Un évaluateur par séquences de visites pour une grammaire attribuée résultant de la méta-composition

Dans un premier temps nous nous intéressons à la méta-composition de deux grammaires fonctionnelles α et β, mais nous pouvons généraliser les résultats de cette section à une séquence de plus de deux grammaires attribuées fonctionnelles. Nous allons construire un évaluateur par séquences de visites pour la grammaire attribuée résultante, γ.

Il n'est pas nécessaire de construire cette grammaire attribuée résultante pour construire l'évaluateur, mais la grammaire attribuée résultante aide à comprendre le mécanisme d'évaluation.

D'autre part, comme l'évaluateur créé évalue bien l'ensemble des attributs de la grammaire attribuée résultante et comme celle-ci est équivalente à la séquence de grammaires attribuées fonctionnelles, nous déduisons que la construction de notre évaluateur est correcte.

Nous nous restreignons pour plus de simplicité aux grammaires attribuées fonctionnelles purement syntaxiques simples[2]. Nous introduirons dans un second temps les conditionnelles, les constructions imbriquées et les calculs sémantiques.

L'idée simple qui nous permet de construire l'évaluateur pour la grammaire attribuée résultante est la suivante :

> *Dans la méta-composition, les règles sémantiques accrochées à chaque production p de β, sont projetées sur les règles sémantiques de α dans lesquelles une construction suivant p est effectuée. Comme les règles sémantiques, les visites de l'évaluateur de β sont attachées aux productions et donc par analogie avec la méta-composition, nous appelons les visites de β en suivant les constructions de α.*

Exemple d'évaluateur pour une séquence de deux grammaires fonctionnelles

Nous donnons ici l'exemple d'un évaluateur construit pour une grammaire attribuée résultante de la méta-composition sans en expliciter la construction.

Je décris d'abord les deux grammaires fonctionnelles α et β et leur composition γ, puis un évaluateur classique par séquence de visites pour β et un évaluateur par séquence de visites pour γ. Je fais remarquer au passage que la grammaire attribuée γ n'est pas de classe FNC.

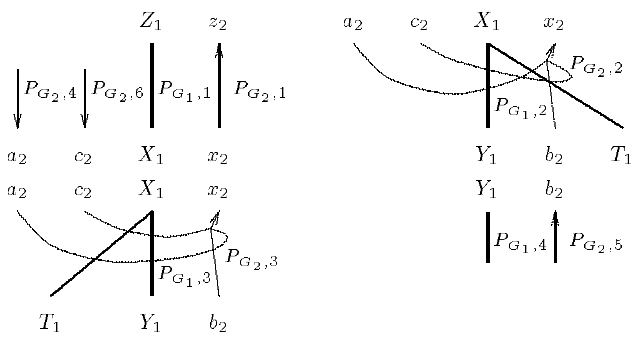
[2]Les règles sémantiques ne sont que des constructions simples ou des règles de copie.

Soit $\alpha : G_1 \rightarrow G_2$:

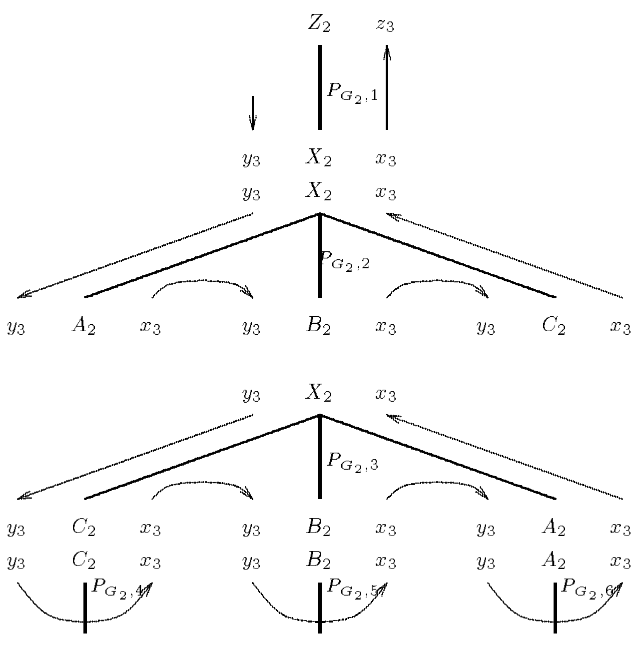
G_1 : $P_{G_1,1}: Z_1 \rightarrow X_1$;
$P_{G_1,2}: X_1 \rightarrow Y_1 T_1$;
$P_{G_1,3}: X_1 \rightarrow T_1 Y_1$;
$P_{G_1,4}: Y_1 \rightarrow$;

G_2 : $P_{G_2,1}: Z_2 \rightarrow X_2$;
$P_{G_2,2}: X_2 \rightarrow A_2 B_2 C_2$;
$P_{G_2,3}: X_2 \rightarrow C_2 B_2 A_2$;
$P_{G_2,4}: A_2 \rightarrow$;
$P_{G_2,5}: B_2 \rightarrow$;
$P_{G_2,6}: C_2 \rightarrow$;

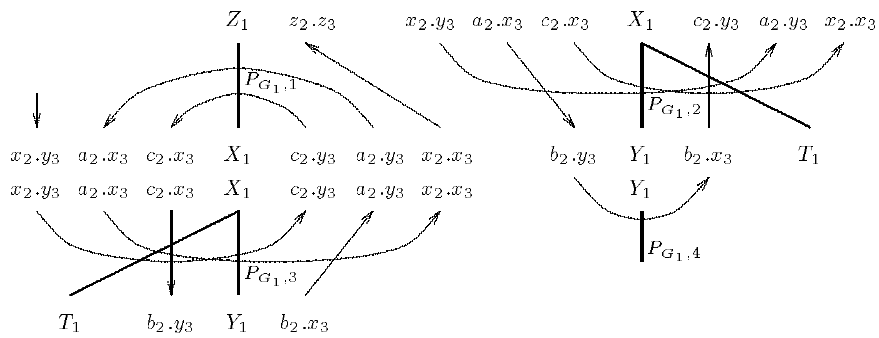
$P_{G_1,1}$:
$Z_1.z2 = P_{G_2,1}(X_1.x_2)$;
$X_1.a2 = P_{G_2,4}()$;
$X_1.c2 = P_{G_2,6}()$;
$P_{G_1,2}$:
$X_1.x_2 = P_{G_2,2}(X_1.a2, Y_1.b_2, X_1.c_2)$;
$P_{G_1,3}$:
$X_1.x_2 = P_{G_2,2}(X_1.c2, Y_1.b_2, X_1.a_2)$;
$P_{G_1,4}$:
$Y_1.b_2 = P_{G_2,5}()$;



Soit $\beta : G_2 \rightarrow S$, grammaire attribuée purement sémantique, dont voici le graphe de dépendances :



Après avoir effectué la méta-composition nous obtenons la grammaire attribuée résultante $\gamma = \alpha \circ \beta : G_1 \to G_3$, dont voici le graphe de dépendances :



Cette grammaire n'est pas FNC, il y a une circularité introduite par les dépendances :

$$X_1.a_2.x_3 \to Y_1.b_2.y_3 \to Y_1.b_2.x_3 \to X_1.a_2.y_3 \to X_1.a_2.x_3$$

L'évaluateur par séquences de visites pour β est de la forme :

- `V-`$P_{G_2,1}$`-`Z_2`-1(a)`
  `E-`X_2`-`y_3 ;
  `V-`X_2`-1(`*fils*`(a,1))` ;
  `E-`Z_2`-`y_3 ;
- `V-`$P_{G_2,4}$`-`A_2`-1(a)`
  `E-`A_2`-`x_3 ;
- `V-`$P_{G_2,5}$`-`B_2`-1(a)`
  `E-`B_2`-`x_3 ;
- `V-`$P_{G_2,6}$`-`C_2`-1(a)`
  `E-`C_2`-`x_3 ;
- `V-`$P_{G_2,2}$`-`X_2`-1(a)`
  `E-`A_2`-`y_3 ;
  `V-`A_2`-1(`*fils*`(a,1))` ;
  `E-`B_2`-`y_3 ;
  `V-`B_2`-1(`*fils*`(a,2))` ;
  `E-`C_2`-`y_3 ;
  `V-`C_2`-1(`*fils*`(a,3))` ;
  `E-`X_2`-`x_3 ;
- `V-`$P_{G_2,3}$`-`X_2`-1(a)`
  `E-`C_2`-`y_3 ;
  `V-`C_2`-1(`*fils*`(a,1))` ;
  `E-`B_2`-`y_3 ;
  `V-`B_2`-1(`*fils*`(a,2))` ;
  `E-`A_2`-`y_3 ;
  `V-`A_2`-1(`*fils*`(a,3))` ;
  `E-`X_2`-`x_3 ;

Nous plongeons maintenant les fonctions d'évaluation dans les constructions de α en les renommant pour obtenir l'évaluateur pour γ.

Par exemple la visite de β sur $P_{G_2,6}$:

$$\texttt{V-}P_{G_2,6}\texttt{-}C_2\texttt{-1(a)}$$

devient en la plongeant sur la règle sémantique :

$$X_1.c_2 = P_{G_2,6}() ;$$

la visite pour γ sur $P_{G_1,1}$:

$$\texttt{V-}P_{G_1,1}\texttt{-}X_1\texttt{-}c_2.\texttt{1(a)}$$

L'appel à la visite numéro 1 sur le non-terminal C_2 de la production $P_{G_2,6}$ est transformé en un appel à la visite numéro 1 sur le non-terminal X_1 de la production $P_{G_1,1}$ en suivant la structure de la règle sémantique définissant $X_1.c_2$.

La visite V-$P_{G_2,6}$-C_2-1 descend dans l'arbre de G_2 mais comme la construction de $P_{G_2,6}$ est retournée par rapport à l'arbre d'entrée de α[3], la nouvelle visite remonte dans l'arbre d'entrée.

Plus précisément les séquences de visites suivent "l'arbre virtuel" qui aurait été construit par α. Ce cheminement est directement codé dans le nom des fonctions d'évaluation.

Les visites de γ sont associées à une production p et à une occurrence d'attribut de α sur p. Le nom de cette occurrence d'attribut apparaît dans le nom de la fonction d'évaluation, X_1-c_2 dans l'exemple précédent. Les visites ne sont plus appelées sur un nœud mais sur un nœud suivant la définition d'une occurrence d'attribut de α. L'évaluateur suit bien l'arbre intermédiaire qui aurait été construit par α.

Deux visites appelées sur un même nœud sont différenciées par un attribut et un numéro de visite.

L'évaluateur pour γ est le suivant :

- V-$P_{G_1,1}$-Z_1-z_2.1(a)
  E-X_1-$x_2.y3$;
  V-X_1-x_2.1(*fils*(a,1)) ;
  E-Z_1-$z_2.z_3$;
- V-$P_{G_1,1}$-X_1-a_2.1(a)
  E-X_1-$a_2.x3$;
- V-$P_{G_1,1}$-X_1-c_2.1(a)
  E-X_1-$c_2.x3$;
- V-$P_{G_1,4}$-Y_1-b_2.1(a)
  E-Y_1-$b_2.x3$;
- V-$P_{G_1,2}$-X_1-x_2.1(a)
  E-X_1-$a_2.y3$;
  V-X_1-a_2.1(a) ;
  E-Y_1-$b_2.y_3$;
  V-Y_1-b_2.1(*fils*(a,1)) ;
  E-X_1-$c_2.y3$;
  V-X_1-c_2.1(a) ;
  E-X_1-$x_2.x_3$;
- V-$P_{G_1,3}$-X_1-x_2.1(a)
  E-X_1-$c_2.y3$;
  V-X_1-c_2.1(a) ;
  E-Y_1-$b_2.y_3$;
  V-Y_1-b_2.1(*fils*(a,1)) ;
  E-X_1-$a_2.y3$;
  V-X_1-a_2.1(a) ;
  E-X_1-$x_2.x_3$;

4.1.3 Algorithme de construction de l'évaluateur

Nous décrivons maintenant de façon précise l'algorithme de construction de ces nouvelles séquences de visites.

Plutôt que de construire totalement l'évaluateur de γ comme dans l'exemple précédent nous construisons ici des filtres qui appellent correctement les visites de β sur l'arbre d'entrée de α.

Cette méthode a l'avantage d'éviter de réécrire totalement les fonctions d'évaluation.

[3] C'est la définition d'un attribut hérité.

Préliminaires

Nous supposons que lors de la méta-composition tous les attributs de $\gamma = \alpha \circ \beta$ sont déclarés et la syntaxe de base G_1 est décrite.

Les filtres ont le nom des fonctions d'évaluation de γ qui auraient été construites si nous avions procédé comme dans l'exemple précédent. Ils sont donc nommés comme des visites "classiques".

L'appel aux fonctions d'évaluation de β par les filtres implique que celles-ci soient paramétrables. En effet une même fonction d'évaluation peut être appelée par deux filtres correspondant à deux contextes différents.

Une fonction d'évaluation de β travaille sur une instance de production de G_2, il faut donc passer à chaque fonction d'évaluation de β une instance de production "virtuelle" sur laquelle l'évaluateur est capable d'accéder aux différentes instances attributs manipulées par la fonction d'évaluation.

Une fonction d'évaluation reçoit donc en paramètres les nœuds "virtuels" de l'instance de production "virtuelle" de G_2 sur laquelle elle travaille.

Nœud virtuel Les nœuds virtuels sont constitués d'un nœud réel u de l'arbre de G_1 et d'un attribut a qu'il porte dans α. Soit $u.a$ un tel nœud virtuel, tous les attributs $b \in \mathcal{A}_\beta(type(a))$ lui sont associés. L'instance de l'attribut b sur le nœud virtuel $u.a$ est notée $u.a.b$. Elle est équivalente à l'instance de l'attribut $a.b$ sur le nœud u pour γ notée également $u.a.b$.

Instance de production courante Il est également nécessaire de passer aux filtres et aux visites de β un autre paramètre, le père de l'instance de production courante, noté `U`. L'instance de production courante correspond à l'instance de production de l'arbre réel de G_1 sur laquelle l'évaluateur est en train de travailler. Elle permet de se déplacer dans l'arbre réel.

Comportement de l'évaluateur

Nous décrivons ici le mécanisme de l'évaluateur.

Lorsqu'une fonction d'évaluation de β appelle une nouvelle visite, cette dernière est appelée sur un nœud virtuel $u.a$.

Afin d'appeler correctement la nouvelle visite de l'évaluateur de β, l'évaluateur doit se déplacer dans l'arbre virtuel, c'est-à-dire trouver l'instance de production virtuelle dont $u.a$ est le père. Pour ce faire, nous procédons comme suit.

Grâce à la partie réelle u de $u.a$ et au père de l'instance de production courante U l'évaluateur, par l'intermédiaire du sélecteur de visite, se déplace dans l'arbre réel.

Ensuite, le nom de l'attribut a de $u.a$ permet à ce dernier de retrouver le bon filtre de l'appeler avec le nouveau père de la production courante U et la production réelle comme paramètres.

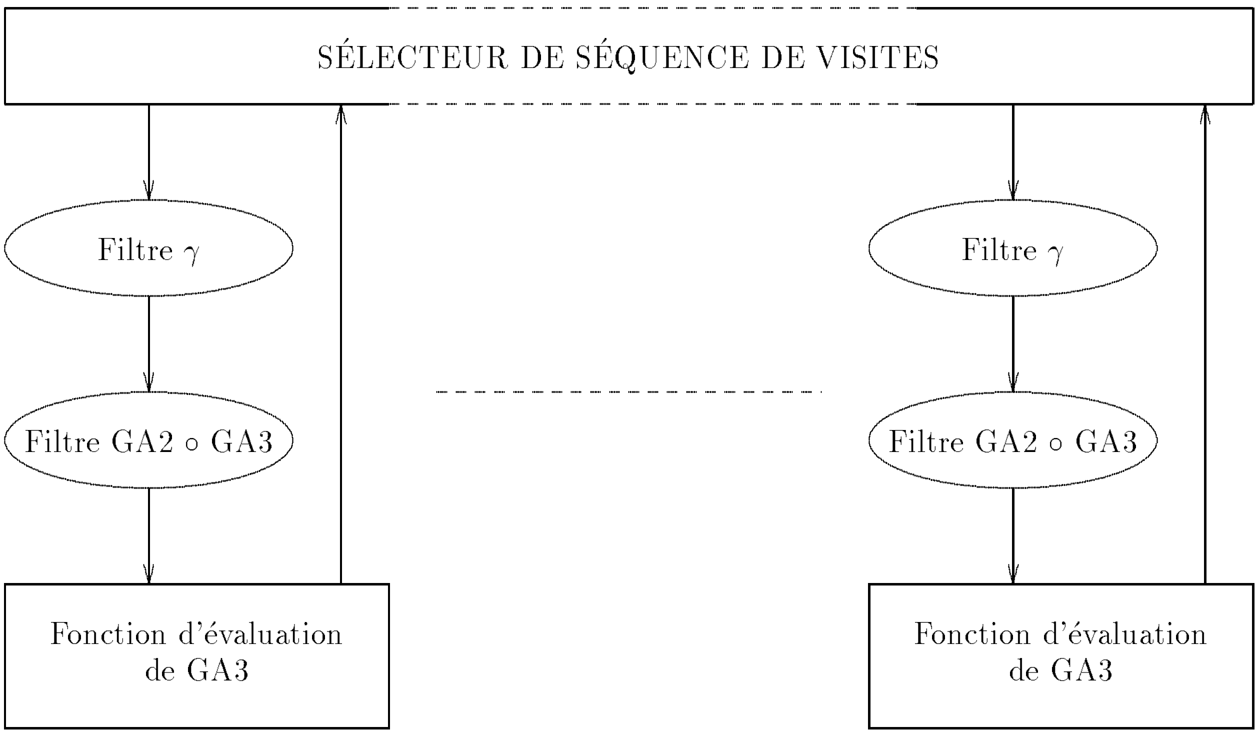


Figure 4.1: Enchaînement d'appels pour l'évaluateur de γ = GA1 $\circ$ GA2 $\circ$ GA3.

Le filtre construit à partir de la production réelle, la nouvelle instance de production virtuelle et appelle finalement la nouvelle visite de β sur cette nouvelle instance de production virtuelle.

L'évaluateur ainsi produit a la même forme qu'un évaluateur classique. La seule différence est que si l'évaluateur a plus d'une fonction d'évaluation associée à un même non-terminal, il les différencie par un numéro de visite mais aussi par un attribut d'appel.

Il est possible d'appliquer l'algorithme qui suit aussi bien sur un évaluateur par séquence de visites que sur un évaluateur produit par la composition des évaluateurs. Je me suis restreint dans la présentation rapide que j'ai donnée précédemment à un évaluateur "Top-Down" mais la construction peut très bien s'appliquer à d'autre types d'évaluateurs, en particulier des évaluateurs de type DNC.

Une idée des appels aux différents filtres de γ = GA1 $\circ$ GA2 $\circ$ GA3 est donnée par la figure 4.1.

Nous allons maintenant décrire de façon précise la construction d'un évaluateur pour $\gamma = \alpha \circ \beta$. Pour cela nous avons besoin de préciser la forme des fonctions d'évaluation de β. Cette forme est aussi celle des filtres de γ.

Forme des fonctions d'évaluation et des filtres

Les fonctions d'évaluation et les filtres s'écrivent de façon générique :

$$\texttt{V-}P_{G_2,i}\texttt{-}pos\texttt{-ID(U,W}_0\texttt{,}\cdots\texttt{,W}_k\texttt{)}$$

où :

- $P_{G_2,i}$ est la production de G_2 sur laquelle cette fonction d'évaluation est appelée ;
- *pos* est la position dans la production $P_{G_2,i}$ de l'occurrence de non-terminal sur laquelle la visite a été appelée ; dans un évaluateur classique on utilise le nom de cette occurrence de non-terminal plutôt que cette position ;
- `ID` est un identificateur qui permet de différencier deux fonctions d'évaluation ayant un même profil, en général un numéro de visite ou un attribut plus un numéro de visite ;
- `U` est le paramètre auquel auquel est passé le nœud père de l'instance de production courante sur laquelle la fonction d'évaluation est appelée ;
- les $\texttt{W}_j$ sont les paramètres auxquels sont passés les nœuds de l'instance de production sur laquelle la fonction d'évaluation est appelée.

Dans les fonctions d'évaluation, les appels à de nouvelles visites sont de la forme :

$$\texttt{V} - \texttt{ID}'(\texttt{U},\texttt{W})$$

où :

- $\texttt{ID}'$ est l'identificateur de la nouvelle visite appelée ;
- `U` est le père de l'instance de production courante ;
- `W` est nœud virtuel sur lequel se fait l'appel.

Maintenant nous réécrivons sous cette nouvelle forme les fonctions d'évaluation de la grammaire attribuée β, présentée précédemment.

- $\texttt{V-}P_{G_2,1}\texttt{-0-1(U,W}_0\texttt{,W}_1\texttt{)}$
  $\texttt{E-}X_2\texttt{-}y_3\texttt{(W}_1\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_1\texttt{)}$;
  $\texttt{E-}Z_2\texttt{-}z_3\texttt{(W}_0\texttt{.}z_3\texttt{)}$
- $\texttt{V-}P_{G_2,4}\texttt{-0-1(U,W}_0\texttt{)}$
  $\texttt{E-}A_2\texttt{-}x_3\texttt{(W}_0\texttt{.}x_3\texttt{)}$;
- $\texttt{V-}P_{G_2,5}\texttt{-0-1(U,W}_0\texttt{)}$
  $\texttt{E-}B_2\texttt{-}x_3\texttt{(W}_0\texttt{.}x_3\texttt{)}$;
- $\texttt{V-}P_{G_2,6}\texttt{-0-1(U,W}_0\texttt{)}$
  $\texttt{E-}C_2\texttt{-}x_3\texttt{(W}_0\texttt{.}x_3\texttt{)}$;
- $\texttt{V-}P_{G_2,2}\texttt{-0-1(U,W}_0\texttt{,W}_1\texttt{,W}_2\texttt{,W}_3\texttt{)}$
  $\texttt{E-}A_2\texttt{-}y_3\texttt{(W}_1\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_1\texttt{)}$;
  $\texttt{E-}B_2\texttt{-}y_3\texttt{(W}_2\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_2\texttt{)}$;
  $\texttt{E-}C_2\texttt{-}y_3\texttt{(W}_3\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_3\texttt{)}$;
  $\texttt{E-}X_2\texttt{-}x_3\texttt{(W}_0\texttt{.}x_3\texttt{)}$;
- $\texttt{V-}P_{G_2,3}\texttt{-0-1(U,W}_0\texttt{,W}_1\texttt{,W}_2\texttt{,W}_3\texttt{)}$
  $\texttt{E-}C_2\texttt{-}y_3\texttt{(W}_1\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_1\texttt{)}$;
  $\texttt{E-}B_2\texttt{-}y_3\texttt{(W}_2\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_2\texttt{)}$;
  $\texttt{E-}A_2\texttt{-}y_3\texttt{(W}_3\texttt{.}y_3\texttt{)}$;
  $\texttt{V-1(U,W}_3\texttt{)}$;
  $\texttt{E-}X_2\texttt{-}x_3\texttt{(W}_0\texttt{.}x_3\texttt{)}$;

Nous décrivons dès maintenant les filtres pour γ produits par la composition des évaluateurs pour que le lecteur puisse les comparer avec l'évaluateur précédent et comprendre le mécanisme du sélecteur de visites que nous présentons ensuite.

- V-$P_{G_1,1}$-0-z_2.1(U,$\mathtt{W}_0$,$\mathtt{W}_1$)
  V-$P_{G_2,1}$-0-1(U,$\mathtt{W}_0$.z_2,$\mathtt{W}_1$.x_2) ;
- V-$P_{G_1,2}$-0-x_2.1(U,$\mathtt{W}_0$,$\mathtt{W}_1$)
  V-$P_{G_2,2}$-0-1(U,$\mathtt{W}_0$.x_2,$\mathtt{W}_0$.a_2,$\mathtt{W}_1$.b_2,$\mathtt{W}_0$.c_2) ;
- V-$P_{G_1,3}$-0-x_2.1(U,$\mathtt{W}_0$,$\mathtt{W}_1$)
  V-$P_{G_2,3}$-0-1(U,$\mathtt{W}_0$.x_2,$\mathtt{W}_0$.c_2,$\mathtt{W}_1$.b_2,$\mathtt{W}_0$.a_2) ;
- V-$P_{G_1,1}$-1-a_2.1(U,$\mathtt{W}_0$,$\mathtt{W}_1$)
  V-$P_{G_2,4}$-0-1(U,$\mathtt{W}_0$.a_2) ;
- V-$P_{G_1,1}$-1-c_2.1(U,$\mathtt{W}_0$,$\mathtt{W}_1$)
  V-$P_{G_2,6}$-0-1(U,$\mathtt{W}_0$.c_2) ;
- V-$P_{G_1,4}$-0-b_2.1(U,$\mathtt{W}_0$)
  V-$P_{G_2,5}$-0-1(U,$\mathtt{W}_0$.b_2) ;

Le sélecteur de visite

Le sélecteur de visite est le mécanisme qui retrouve le bon filtre à appeler et les bons arguments à lui passer lorsqu'une fonction d'évaluation de β appelle une nouvelle visite.

Supposons qu'une nouvelle visite soit appelée. L'appel au sélecteur de visite est de la forme :

`V-ID(U,nœud.virt)`

où `nœud` est le nœud réel associé au nœud virtuel `nœud.virt`.

Si `nœud` est le père de l'instance de production courante, il faut remonter dans l'arbre, sinon `nœud` est un fils et il faut descendre dans l'arbre. Le sélecteur de visite retrouve le nom du filtre à appeler grâce à :

1. la partie attribut `virt` de `nœud.virt` et à `ID` de l'appel à la nouvelle visite ;
2. la position de `nœud` dans la nouvelle instance de production courante.

Le sélecteur est alors de la forme :

Fonction `V-ID(U,nœud.virt)`

Si `nœud` = `U` **alors**
  `U` $\leftarrow$ *père*(`nœud`) ;
  pos $\leftarrow$ *position*(`nœud`, `U`) ;
Sinon
  `U` $\leftarrow$ `nœud` ;
  pos $\leftarrow$ 0 ;
Fin si
Dans le cas où *label*(`U`) **est**
  $\vdots$
  P_{G_1,i_a} : $X_0 \to X_1 \cdots X_n$ **faire**

V-P_{G_1,i_a}-pos-virt.ID (U,instance production courante)
$\vdots$

Fin dans le cas où

Production copie

Afin de décrire plus simplement la construction des filtres nous introduisons la notion de production copie.

Définition 17 (Production Copie) *J'appelle production copie d'une grammaire attribuée α toute production $P_{G_1,\mathcal{Y}}$: $Y_0 \rightarrow Y_1$*[4] *où $\mathcal{Y}$ est un non-terminal de α. A celle-ci nous associons un ensemble de règles sémantiques $E_{P_{G_1,\mathcal{Y}}}$. Pour chaque attribut de $\mathcal{Y}$ on construit la règle sémantique :*

- $r_{P_{G_1,\mathcal{Y}},a,0}$: $Y_0.a = Y_1.a$, *si* $a \in \mathcal{S}(\mathcal{Y})$ *;*
- $r_{P_{G_1,\mathcal{Y}},a,1}$: $Y_1.a = Y_0.a$, *si* $a \in \mathcal{H}(\mathcal{Y})$*.*

La construction des fonctions d'évaluation associées à ces productions est simple (cf. Figure 4.2).

Pour chacun des non-terminaux de β, nous construisons un production copie associée. Nous construisons également les fonctions d'évaluation sur celles-ci.

Ces fonctions d'évaluation seront appelées sur les règles de copie de α. Elles correspondent aux règles sémantiques construites sur les règles de copie dans l'algorithme de méta-composition de Ganzinger et Giegerich.

Construction des filtres pour $\gamma = \alpha \circ \beta$

Toutes ces notions étant précisées, nous pouvons décrire la construction de l'évaluateur. Celle-ci se divise en quatre étapes :

1. Soit la règle sémantique de α sur P_{G_1,i_a} : $X_0 \rightarrow X_1 \cdots X_n$:

   $$r_{P_{G_1,i_a},y_0,f(0)} : X_{f(0)}.y_0 = P_{G_2,i_b}(X_{f(1)}.y_1, \cdots, X_{f(m)}.y_m)$$

   où P_{G_2,i_b} est une production de G_2, $X_{f(i)}.y_i$ une occurrence d'attribut de type syntaxique ou de type sémantique de α. Pour chaque fonction d'évaluation de la forme :

   V-P_{G_2,i_b}-pos-ID(U,W$_0$,$\cdots$,W$_m$)

   nous construisons un filtre pour γ de la forme :

   V-P_{G_1,i_a}-$f(pos)$-y_{pos}.ID(U,W$_0$,$\cdots$,W$_n$)

[4] Nous supposons que cette production n'existe pas déjà, quitte à un renommage.

Soit production copie $P_{G_1,\mathcal{Y}} : Y_0 \rightarrow Y_1$, avec :

$\mathcal{R}(\mathcal{Y}) = \{(a,c),(c,b)\}$;

$P_{G_1,\mathcal{Y}} : Y_0 \rightarrow Y_1$;
  $Y_0.a = Y_1.a$;
  $Y_0.b = Y_1.b$;
  $Y_1.c = Y_0.c$;

Nous avons les fonctions d'évaluation suivantes :

V-$P_{G_1,\mathcal{Y}}$-0-1 (U,W_0,W_1)
  V-1 (U,W_1) ;
  $W_0.a = W_1.a$;
V-G-py-0-2 (U,W_0,W_1)
  $W_1.c = W_0.c$;
  V-2 (U,W_1) ;
  $W_0.b = W_1.b$;
V-py-1-1 (U,W_0,W_1)
  $W_0.a = W_1.a$;
  V-1 (U,W_0) ;
  $W_1.c = W_0.c$;
V-py-1-2 (U,W_0,W_1)
  $W_0.b = W_1.b$;
  V-2 (U,W_0)

Figure 4.2: Exemple de production copie et de fonctions d'évaluation associées

et tel que :

$$\texttt{V-}P_{G_1,i_a}\texttt{-}f(pos)\texttt{-}y_{pos}\texttt{.ID(}U\texttt{,}u_0\texttt{,}\cdots\texttt{,}u_n\texttt{)=V-}P_{G_2,i_b}\texttt{-}pos\texttt{-ID(}U\texttt{,}u_{f(0)}.y_0\texttt{,}\cdots\texttt{,}u_{f(m)}.y_m\texttt{)}$$

2. Soit la règle sémantique de α sur $P_{G_1,i_a} : X_0 \to X_1 \cdots X_n$:

$$r_{P_{g_1,i_a},y_0,f(0)} : X_{f(0)}.y_0 = X_{f(1)}.y_1$$

   où les $X_{f(i)}.y_i$ sont des occurrences d'attributs d'un même type $\mathcal{Y}$. Pour chaque fonction d'évaluation sur la production copie $P_{G_2,\mathcal{Y}}$ de la forme :

$$\texttt{V-}P_{G_2,\mathcal{Y}}\texttt{-}pos\texttt{-ID(U,W}_0\texttt{,W}_1\texttt{)}$$

   nous construisons un filtre de la forme :

$$\texttt{V-}P_{G_1,i_a}\texttt{-}f(pos)\texttt{-}y_0\texttt{.ID(U,W}_0\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

   et tel que :

$$\texttt{V-}P_{G_1,i_a}\texttt{-}f(pos)\texttt{-}y_0\texttt{.ID(}U\texttt{,}u_0\texttt{,}\cdots\texttt{,}u_n\texttt{)=V-}P_{G_2,\mathcal{Y}}\texttt{-}pos\texttt{-ID(}U\texttt{,}u_{f(0)}.y_0\texttt{,}u_{f(1)}.y_1\texttt{)}$$

3. Soit une production copie $P_{G_1,\mathcal{Y}} : Y_0 \to Y_1$ et $x \in \mathcal{S}(\mathcal{Y})$ un attribut de type $\mathcal{X}$. Pour chaque fonction d'évaluation de la forme :

$$\texttt{V-}P_{G_2,\mathcal{X}}\texttt{-}pos\texttt{-ID(U,W}_0\texttt{,W}_1\texttt{)}$$

   nous construisons un filtre de la forme :

$$\texttt{V-}P_{G_1,\mathcal{Y}}\texttt{-}pos\texttt{-}x\texttt{.ID(U,W}_0\texttt{,W}_1\texttt{)}$$

   et tel que :

$$\texttt{V-}P_{G_1,\mathcal{Y}}\texttt{-}pos\texttt{-}x\texttt{.ID (}U\texttt{,}u_0\texttt{,}u_1\texttt{)=V-}P_{G_2,\mathcal{X}}\texttt{-}pos\texttt{-ID(}U\texttt{,}u_0.x\texttt{,}u_1.x\texttt{)}$$

4. Soit une production copie $P_{G_1,\mathcal{Y}} : Y_0 \to Y_1$ et $x \in \mathcal{H}(\mathcal{Y})$ un attribut de type $\mathcal{X}$. Pour chaque fonction d'évaluation de la forme :

$$\texttt{V-}P_{G_2,\mathcal{X}}\texttt{-}pos\texttt{-ID(U,W}_0\texttt{,W}_1\texttt{)}$$

   Nous construisons un filtre de la forme :

$$\texttt{V-}P_{G_1,\mathcal{Y}}\texttt{-}pos'\texttt{-}x\texttt{.ID(U,W}_0\texttt{,W}_1\texttt{)}$$

   et tel que :

$$pos' = \neg pos$$
$$\text{et}$$
$$\texttt{V-}P_{G_1,\mathcal{Y}}\texttt{-}pos'\texttt{-}x\texttt{.ID(}U\texttt{,}u_0\texttt{,}u_1\texttt{)=V-}P_{G_2,\mathcal{X}}\texttt{-}pos\texttt{-ID(}U\texttt{,}u_1.x\texttt{,}u_0.x\texttt{)}$$

Remarques générales Les constructions 1 et 2 sont nécessaires pour la construction de l'évaluateur de γ.

Les constructions 3 et 4 ne sont nécessaires que pour pouvoir composer γ avec une autre grammaire attribuée fonctionnelle. En effet, elles construisent les fonctions d'évaluations sur les productions copie de γ qui ne seront jamais appelées par l'évaluateur de γ. C'est uniquement lorsque γ est composée avec une autre grammaire attribuée fonctionnelle que ces fonctions d'évaluations sont susceptibles d'être appelées par un filtre de la grammaire résultante construit à l'étape 2.

Il est possible de coder statiquement les différents appels aux filtres dans le sélecteur, l'évaluateur évite ainsi les appels successifs aux différents filtres.

Lorsque β est modifiée, si l'ensemble des noms de fonctions d'évaluation de β ne change pas, les filtres construit sont encore valides. Il n'est alors pas nécessaire de recompiler toute l'application mais seulement β.

4.2 Ajout des conditionnelles

Nous supposons toujours qu'il n'y a pas de calculs sémantiques dans les grammaires attribuées. Les conditions sont donc des valeurs de terminaux.

Les conditionnelles de β ne nous gênent pas puisqu'elles sont directement codées dans les séquences de visites. Les seules conditionnelles qui nous intéressent sont celles de α. Nous nous bornerons aux conditionnelles sur une production $P_{G_1,i_a} : X_0 \rightarrow X_1 \cdots X_n$ de α de la forme :

$$
\begin{aligned}
r_{P_{G_1,i_a},c,i} : X_i.c = \; & \textbf{Si } T.t \textbf{ alors} \\
& \qquad p_{G_2,1}(X_{f(1).a_1}, \cdots, X_{f(U)}.a_p)\,; \\
& \textbf{Sinon} \\
& \qquad p_{G_2,2}(X_{g(1).b_1}, \cdots, X_{g(q)}.b_q)\,;
\end{aligned}
$$

En effet toutes les autres conditionnelles peuvent se décomposer sous cette forme. D'autre part, s'il y a une règle de copie au lieu d'une construction dans une des branches du **Si**, il est facile de généraliser.

Nous avons alors deux types de constructions :

1. il existe dans β pour $p_{G_2,1}$ et $p_{G_2,2}$ des fonctions d'évaluation “équivalentes” :

$$\texttt{V-}p_{G_2,1}\texttt{-}pos_1\texttt{-ID(U,W}_0\texttt{,}\cdots\texttt{,W}_p\texttt{)}$$
et
$$\texttt{V-}p_{G_2,2}\texttt{-}pos_2\texttt{-ID(U,W}_0\texttt{,}\cdots\texttt{,W}_q\texttt{)}$$

c'est-à-dire, telles que :

$$pos_1 = pos_2 = 0$$
ou bien
$X_{f(pos_1)}.a_{pos_1}$ représente la même occurrence d'attribut que $X_{g(pos_2)}.b_{pos_2}$[5]

Alors nous construisons un filtre de la forme :

$$\texttt{V-}p_{G_1,i_a}\texttt{-}f(pos_1)\texttt{-}a_{pos_1}\texttt{.ID(U,W}_1\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

et tel que :

$$\texttt{V}-p_{G_1,i_a}-f(pos_1)-a_{pos_1}.\texttt{ID}(U, u_0, \cdots, u_n) = \begin{cases} \textbf{Si } T.t \textbf{ alors} \\ \quad \texttt{V}-p_{G_2,1}-pos_1-\texttt{ID}(U, u_i.c, u_{f(1)}.a_1, \cdots, u_{f(U)}.a_p) \\ \textbf{Sinon} \\ \quad \texttt{V}-p_{G_2,2}-pos_2-\texttt{ID}(U, u_i.c, u_{g(1)}.b_1, \cdots, u_{g(q)}.b_q) \end{cases}$$

2. il n'existe pas dans β de fonction d'évaluation équivalente à $\texttt{V-}p_{G_2,1}\texttt{-}pos_1\texttt{-ID}$ pour $p_{G_2,2}$ au sens précédent. Nous construisons alors un filtre pour γ de la forme :

$$\texttt{V-}p_{G_1,i_a}\texttt{-}f(pos_1)\texttt{-}a_{pos_1}\texttt{.ID(U,W}_0\texttt{,W}_1\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

et tel que :

$$\begin{aligned} &\texttt{V}-p_{G_1,i_a}-f(pos_1)-a_{pos_1}.\texttt{ID}(U, u_0, \cdots, u_n) = \\ &\qquad \texttt{V}-p_{G_2,1}-pos_1-\texttt{ID}(U, u_i.c, u_{f(1)}.a_1, \cdots, u_{f(U)}.a_p) \end{aligned}$$

3. il n'existe pas dans β de fonction d'évaluation équivalente à $\texttt{V-}p_{G_2,2}\texttt{-}pos_2\texttt{-ID}$ pour $p_{G_2,1}$ au sens précédent. Nous construisons alors un filtre pour γ de la forme :

$$\texttt{V-}p_{G_1,i_a}\texttt{-}g(pos_2)\texttt{-}a_{pos_2}\texttt{.ID(U,W}_0\texttt{,W}_1\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

avec :

$$\begin{aligned} &\texttt{V}-p_{G_1,i_a}-g(pos_2)-a_{pos_2}.\texttt{ID}(U, u_0, \cdots, u_n) = \\ &\qquad \texttt{V}-p_{G_2,2}-pos_2-\texttt{ID}(U, u_i.c, u_{g(1)}.a_1, \cdots, u_{g(q)}.a_q) \end{aligned}$$

[5]Cette occurrence d'attribut est un dag dans α.

4.2.1 Constructions imbriquées et attributs locaux

Pour les constructions imbriquées, comme le proposent Ganzinger et Giegerich, nous allons couper celles-ci en constructions élémentaires et utiliser des attributs locaux pour faire le lien entre elles. Pour plus de simplicité, nous associons les attributs locaux à l'occurrence de non-terminal père de la production, nous les notons de façon générique $X_0.loc$. Ce qui nous donne dans le cas de la règle sémantique suivante :

$$r_{p,a,f(0)} : X_{f(0)}.a = P_1(X_{f(1)}.b, P_2())$$

la décomposition :

$$\begin{aligned} r_{p,a,f(0)} &: X_{f(0)}.a = P_1(X_{f(1)}.b, X_0.loc) \\ r_{p,a,0} &: X_0.loc = P_2() \end{aligned}$$

Les attributs locaux ayant la même forme que des occurrences d'attributs classiques nous construisons les filtres pour γ sur ces nouvelles règles sémantiques exactement de la même façon que pour les règles sémantiques classiques. En particulier, s'il y a sur une production $P_{G_1,i_a} : X_0 \to X_1 \cdots X_n$ de α une règle sémantique de la forme :

$$r_{P_{G_1,i_a},loc_1,0} : X_0.loc_1 = P_{G_2,i_b}(X_{f(1)}.y_1, \cdots, X_{f(pos)}.loc_2, \cdots, X_{f(m)}.y_m)^6$$

avec les y_i qui ne sont pas des attributs locaux.

- Pour chaque fonction d'évaluation de β de la forme :

  $$\texttt{V-}P_{G_2,i_b}\texttt{-0-ID (U,W}_0\texttt{,}\cdots\texttt{,W}_m\texttt{)}$$

  nous construisons dans γ un filtre de la forme :

  $$\texttt{V-}P_{G_1,i_a}\texttt{-0-}loc_1\texttt{.ID(U,W}_0\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

  et tel que :

  $$\begin{aligned} &\texttt{V}-P_{G_1,i_a}-0-loc_1.\texttt{ID}(U, u_0, \cdots, u_n) = \\ &\quad \texttt{V}-P_{G_2,i_b}-0-\texttt{ID}(U, u_{f(0)}.loc_1, u_{f(1)}.y_1, \cdots, u_{f(pos)}.loc_2, \cdots, u_{f(m)}.y_m) \end{aligned}$$

- Pour chaque fonction d'évaluation de β de la forme :

  $$\texttt{V-}P_{G_2,i_b}\texttt{-}pos\texttt{-ID (U,W}_0\texttt{,}\cdots\texttt{,W}_m\texttt{)}$$

  nous construisons dans γ un filtre de la forme :

  $$\texttt{V-}P_{G_1,i_a}\texttt{-0-}loc_2\texttt{.ID(U,W}_0\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

[6]Comme tous les attributs locaux sont accrochés au père de la production $f(pos) = 0$.

et tel que :

$$\begin{aligned}&\texttt{V}-P_{G_1,i_a}-0-loc_2.\texttt{ID}(P, u_0, \cdots, u_n) = \\ &\quad \texttt{V}-P_{G_2,i_b}-f(pos)-\texttt{ID}(P, u_{f(0)}.loc_1, u_{f(1)}.y_1, \cdots, u_{f(pos)}.loc_2, \cdots, u_{f(m)}.y_m)\end{aligned}$$

- Pour chaque fonction d'évaluation de β de la forme :

$$\texttt{V-}P_{G_2,i_b}\texttt{-}i\texttt{-ID (U,W}_0\texttt{,}\cdots\texttt{,W}_m\texttt{)}$$

avec :

$$i \neq 0 \text{ et } i \neq pos$$

nous construisons dans γ un filtre de la forme :

$$\texttt{V-}P_{G_1,i_a}\texttt{-}f(i)\texttt{-}y_i\texttt{.ID(U,W}_0\texttt{,}\cdots\texttt{,W}_n\texttt{)}$$

et tel que :

$$\begin{aligned}&\texttt{V}-P_{G_1,i_a}-f(i)-y_i.\texttt{ID}(U, u_0, \cdots, u_n) = \\ &\quad \texttt{V}-P_{G_2,i_b}-i-\texttt{ID}(U, u_{f(0)}.loc_1, u_{f(1)}.y_1, \cdots, u_{f(pos)}.loc_2, \cdots, u_{f(m)}.y_m)\end{aligned}$$

Le nom de l'attribut local n'apparaît dans l'identificateur du filtre que si la fonction d'évaluation de β est appelée dans l'arbre intermédiaire sur cet attribut local, sinon il apparaît uniquement dans le nom d'un des nœuds virtuels. Après plusieurs compositions d'évaluateurs successives l'identificateur et les nœuds virtuels peuvent contenir plusieurs attributs locaux.

Ces attributs locaux vont apparaître dans les appels aux nouvelles séquences de visite puisque ceux-ci sont passés aux fonctions d'évaluation de β comme des nœuds virtuels classiques. Il faut donc modifier en conséquence le sélecteur de visite.

Chaque fois que le nœud virtuel d'appel est "local" il faut rester sur la même production. Pour cela, nous supposons avoir une fonction *local* qui détermine si le nœud virtuel provient d'un attribut local ou non.

Il faut retrouver ensuite la position du nœud d'appel. Après une seule composition des évaluateurs la position est toujours 0 puisque l'attribut local est attaché au père de la production. Par contre après plusieurs compositions l'attribut local peut se retrouver sur un des fils.

Le sélecteur de visite devient alors :

Fonction `V-ID(U,nœud.virt)`

Si *local*(`nœud.virt`) **alors**
  Si `nœud` = `U` **alors**
    pos ← 0 ;
  Sinon
    pos ← *position*(`nœud`, `U`) ;
  Fin si
Sinon
  Si `nœud` = `U` **alors**
    `U` ← *père*(`nœud`) ;
    pos ← *position*(`nœud`,`U`) ;
  Sinon
    `P` ← *label*(`nœud`) ;
    pos ← 0 ;
  Fin si
Fin si
Dans le cas où `U` **est**
    ⋮
  P_{G_1,i_a} : $X_0 \rightarrow X_1 \cdots X_n$ **faire**
    `V-`P_{G_1,i_a}`-`*pos*`-virt.ID` (`U`,instance de production courante) ;
    ⋮
Fin dans le cas où

Remarques générales

Nous remarquons qu'il n'est pas nécessaire de construire la grammaire attribuée γ, il nous suffit de déclarer l'ensemble de ces attributs.

Nous pouvons ainsi relaxer la contrainte concernant la construction d'une forêt de la méta-composition. Cette contrainte provenait du fait que la méta-composition pouvait générer dans ces cas-là une grammaire attribuée où certaines occurrences d'attributs de sortie d'une production n'étaient pas définies. Or, d'après la définition des grammaires attribuées, ceci n'est pas acceptable. Par contre d'un point de vue calculatoire si nous nous bornons à calculer la valeur de l'attribut de la racine, nous sommes sûrs que le calcul de la valeur d'instances de ces occurrences d'attributs ne sera jamais nécessaire. Cette contrainte peut donc être relaxée.

4.2.2 La partie sémantique

Dans chacune des grammaires attribuées de la séquence, il est possible de séparer la partie sémantique de la partie syntaxique.

Nous construisons un évaluateur pour la partie purement sémantique qui peut être appelé depuis la racine (nous détaillons une telle construction dans la section 7.2.3).

Pour cela nous construisons un ensemble de dépendances fictives partant de la racine.

Pour composer α avec β, nous supposons avoir un évaluateur par séquence de visites pour β. Nous effectuons la composition des évaluateurs entre la partie syntaxique α et l'évaluateur de β. Nous construisons ensuite l'évaluateur pour la partie purement sémantique de α. Finalement, nous ajoutons l'appel à l'évaluateur de la partie sémantique au début de la première séquence de visites.

Une autre possibilité est de séparer la grammaire attribuée α en deux grammaires attribuées incomplètes. Nous parlerons plus loin de cette possibilité

4.3 Les Filtres de Couplage

Dans cette section nous allons présenter de façon plus formelle les notions présentées précédemment. Nous allons également pousser un peu plus loin la composition des évaluateurs afin de permettre une véritable compilation séparée. L'idée est de ne plus construire chaque filtre en fonction de la forme de l'évaluateur de la grammaire attribuée suivante, mais en fonction de l'ensemble des formes possibles que peuvent prendre les évaluateurs des grammaires attribuées qui peuvent suivre cette grammaire dans une séquence.

Ici encore pour plus de simplicité, je me place dans le cadre des grammaires attribuées fonctionnelles purement syntaxique simples.

L'idée est la suivante, nous construisons un évaluateur un peu particulier pour la dernière grammaire attribuée de la séquence, appelé évaluateur paramétrable. Quand celui-ci a besoin d'un nouveau morceau d'arbre il appelle un sélecteur de visite, noté SV qui fait le déplacement dans l'arbre réel et renvoie l'instance de production courante. Cette instance de production est alors passée aux différents filtres de couplages, notés FC^i associés à chacune des grammaires attribuées de la séquence, pour donner une instance de production pour la grammaire d'entrée de l'évaluateur paramétrable qui peut alors poursuivre son évaluation sur ce nouveau morceau d'arbre.

Je vais introduire la notion formelle de séquence de grammaires attribuées fonctionnelles qui me sera utile dans les définitions suivantes.

Définition 18 (Séquence de grammaires attribuées fonctionnelles) *Une séquence de grammaires attribuées fonctionnelles, notée* $\Sigma_i^j = G^i \overset{AG^i}{\rightarrow} G^{i+1} \cdots G^j \overset{AG^j}{\rightarrow} G^{j+1}$ *est une suite de grammaires attribuées fonctionnelles* AG^k *telles que :* $AG^k : G^k \rightarrow G^{k+1}$ *pour tout* $k \in [i, ..j]$.

Comme nous l'avons vu précédemment nous avons besoin de la notion de nœud sur l'arbre intermédiaire qui aurait était construit, appelé nœud virtuel.

Définition 19 (Nœud Virtuel) *Étant donnée une séquence* Σ_1^j *on définit un nœud virtuel, noté* v *comme une séquence* $v = u.a^1.\cdots.a^{j-1}$ *et son label, noté* $label(v)$ *tel que :*

1. *si* $j = 1$ *alors* $v = u$ *et* $label(v) = label(u)$ *avec* u *nœud pour* G^1 *et donc* $label(u) \in N^1 \cup T^1$ *;*

2. *si* $j > 1$ *alors* $v = v'.a^{j-1}$ *et* $label(v) = type(a^{j-1})$ *avec :*
   - v' *est un nœud virtuel pour la séquence* Σ_1^{j-1} *;*
   - a^{j-1} *est un attribut sur* $label(v')$ *dans la grammaire attribuée* AG^{j-1}*.*

Dans les sections précédentes nous passions un ensemble de nœuds virtuels en paramètres des différents filtres, comme ceux-ci appartiennent à une même instance de production de l'arbre virtuel nous introduisons la notion d'instance de production virtuelle.

Définition 20 (Instance Virtuelle de Production) *Étant donnée une séquence* Σ_1^j *et une production* $p : X_0 \rightarrow X_1 \cdots X_n$ *de la grammaire* G^j*, une instance de production virtuelle pour* p *est de la forme* $V(p) = v_0 \rightarrow v_1 \cdots v_n$*, avec pour chaque* $k \in [0..n]$ *:*

- v_k *est un nœud virtuel pour la séquence* Σ_1^j *;*
- $label(v_k) = X_k$*.*

Nous présentons maintenant l'étape élémentaire, appelée construction virtuelle, qui transforme une instance de production virtuelle, pour la grammaire d'entrée d'une grammaire attribuée fonctionnelle, en une nouvelle instance de production virtuelle pour la grammaire de sortie, conformément à la définition d'une instance d'attribut dans la grammaire attribuée fonctionnelle. Lorsque nous regroupons toutes les constructions virtuelles possibles d'une même grammaire attribuée, nous obtenons le filtre de couplage.

Définition 21 (Construction Virtuelle) *Soit* $AG : G \rightarrow G'$ *une grammaire attribuée fonctionnelle faisant partie d'une séquence* Σ*. On appelle construction virtuelle sur une production* $p : X_0 \rightarrow X_1 \cdots X_n$ *suivant une occurrence d'attribut* $X_j.a$ *de cette production, une fonction qui prend comme paramètres une instance virtuelle de production pour* p*, la position* j *dans* p *du non-terminal qui porte l'occurrence d'attribut* $X_j.a$ *et renvoie une nouvelle instance de production et une position dans celle-ci. On la note :* $< V(p), j > \underset{a}{\Longrightarrow} < V(p'), j' >$*. Elle est définie de la façon suivante :*

1. *soit* f_{p,y^0,Y^0} *l'unique règle sémantique sur* p *où l'occurrence d'attribut* $X_j.a$ *apparaît ; celle-ci est de la forme* $Y^0.y^0 = p'(Y^1.y^1, \ldots, Y^l.y^l)$ *où* $Y^{j'}.y^{j'} = X_j.a$ *avec* $0 \leq j' \leq l$ *et* p' *production de la grammaire* G' *;*

2. $V(p') : w_0 \rightarrow w_1 \cdots w_l$ *avec* $w_s = v_k.y^s$ *si* $Y^s = X_k$ *pour tout* $s \in \{0, \cdots, l\}$*.*

Comme nous avons supposé que chaque occurrence d'attribut était associée à une unique règle sémantique, cette construction est correctement définie.

Pour diriger la suite de constructions virtuelles, nous avons besoin de la notion d'attribut virtuel qui était codée précédemment dans le nom des filtres. Un attribut virtuel pour une séquence donnée est composé d'un liste d'attributs appartenant à chacune des grammaires attribuées fonctionnelles de la séquence.

Définition 22 (Attribut Virtuel) *Étant donnée une séquence Σ_j^m, on définit un attribut virtuel, noté $v(a^j)$, comme une séquence d'attributs de la forme $v(a^j) = a^j.\cdots.a^{m-1}$ et telle que :*

1. *si $j = m$ alors on a qu'un seul attribut virtuel possible $v(\epsilon) = \epsilon$ (séquence vide) ;*
2. *si $j < m$ alors $v(a^j) = a^j.v(a^{j+1})$ où $v(a^{j+1})$ est un attribut virtuel pour la séquence Σ_{j+1}^m et a^{j+1} est soit ϵ, soit un attribut de la grammaire attribuée AG^{j+1} tel que $a^{j+1} \in A^{j+1}(type(a^j))$.*

Cet attribut virtuel est donné au début d'une séquence par le sélecteur de visite que nous présentons plus précisément plus loin.

Nous présentons maintenant le filtre de couplage complet pour une grammaire attribuée fonctionnelle dans une séquence quelconque. Il est important de remarquer que toutes les définitions sont complètement indépendantes de la séquence.

Le filtre de couplage prend en entrée une instance virtuelle de production, un attribut virtuel et une position dans la production ; l'attribut virtuel et la position permettent d'appliquer la bonne construction virtuelle sur l'instance virtuelle de production. Le filtre de couplage renvoie un attribut virtuel pour le reste de la séquence, une nouvelle instance virtuelle de production et une position dans cette dernière.

Définition 23 (Filtre de Couplage) *Soit $AG : G \rightarrow G'$ une grammaire attribuée fonctionnelle, le filtre de couplage de AG, noté FC, est un ensemble de transitions virtuelles ayant la définition suivante :*

1. *pour chaque production $p : X_0 \rightarrow X_1 \cdots X_n$,*
2. *pour chaque règle sémantique $f_{p,y^0,Y^0} : Y^0.y^0 = p'(Y^1.y^1, \ldots, Y^l, y^l)$ sur p,*
3. *pour chaque occurrence d'attribut $Y^{j'}.y^{j'} = X_j.a, 0 \leq j' \leq l$ de cette règle sémantique ;*

on crée une transition virtuelle $< U, V(p), v(a), j > \Longrightarrow < U, V(p'), v(a'), j' >$ où :

U est le nœud père de l'instance de production courante de l'arbre réel ;

$V(p)$ est une instance de production virtuelle pour une séquence de grammaires attribuées fonctionnelles se terminant par AG ;

$v(a)$ est un attribut d'une séquence de grammaires attribuées fonctionnelles commençant par AG ;

j est une position dans p ;

et telle que :

- *$v(a) = a.v(a')$;*
- *$< V(p), j > \underset{a}{\Longrightarrow} < V(p'), j' >$.*

Nous allons présenter maintenant l'évaluateur paramétrable de la dernière grammaire attribuée de la séquence. Il est possible d'utiliser n'importe quel type d'évaluateur paramétrable pour compléter notre construction de l'évaluateur global. Nous nous bornons ici à l'utilisation d'un évaluateur par séquences de visites. L'évaluateur paramétrable se construit de façon presque identique de l'évaluateur classique, c'est pourquoi nous nous bornons à une présentation informelle de cet évaluateur.

Évaluateur Paramétrable à séquence de visites L'évaluateur paramétrable est constitué de visites paramétrables associées à chaque numéro de séquence de visites. Elles sont de la forme :

`V`-num($< U, V(p), \epsilon, j >$)
  case $p \in P$:
    case $j \in [0..n]$:
      $\vdots$
      V-num-*p*-*j*$(U, V(p))$;
      $\vdots$
    End case
  End case

Elles se divisent en fonctions associées à chacune des positions de chacune des productions p, celles-ci sont de la forme :

`V`-num-*p*-*j*$(U, V(p))$
    $\vdots$
  `E`-att(v_i) ;
    $\vdots$
  `V`-num$(\Sigma(U, v_j))$
    $\vdots$

Elles sont constituées de fonctions `E`-att qui évaluent les attributs et d'appel à de nouvelles visites `V`-num.

La fonction Σ construit à partir du nœud virtuel d'appel le nouveau quadruplet $< U, V(p), \epsilon, j >$. Cette fonction s'écrit en fonction des différents filtres de couplages et le sélecteur de visite :
$\Sigma(U, v) = FC^{m-1}(\cdots(FC^1(SV(U, v))))$

Pour compléter l'évaluateur global, nous devons introduire le sélecteur de visite SV qui retrouve, à partir d'un nœud d'appel, la nouvelle instance de production de travail dans l'arbre réel et donne l'attribut virtuel qui va guider les constructions virtuelles de la séquence.

Définition 24 (Sélecteur de visite) *Étant donné un évaluateur paramétrable basé sur des séquences de visites, pour une grammaire attribuée AG qui termine une séquence Σ, on définit un sélecteur de visite noté VS tel que :*

1. *pour un nœud virtuel d'appel $v = u.v(a)$ de la séquence Σ ;*
2. *pour le nœud père U de l'instance de production courante ;*

on construit le triplet $< V(p), v(a), j >$ et le nouveau nœud père de l'instance de production courante U où :

- *si le nœud d'appel u est égal à U alors on remonte dans l'arbre et $U \leftarrow$ `père`(u) d'autre part la position du nœud d'appel j par rapport à U est maintenant telle que $u = Pos(U, j)$;*
- *si le nœud appel u n'est pas égal à U alors on descend dans l'arbre et $U \leftarrow u$ d'autre part la position du nœud d'appel j est maintenant 0 ;*
- *$V(p) : U \rightarrow Pos(U, 1) \cdots Pos(U, n)$ avec $p = prod(U)$*

Il nous reste maintenant à décrire le point de départ de l'évaluateur global.

Point de départ L'évaluation commence par l'appel à la visite `V`-1 dans l'évaluateur paramétrable avec :

U nœud racine de l'arbre d'entrée ;

$v_i = U.z^1. \cdots .z^{m-1}$ où les z^i sont les attributs associés aux différentes racines des grammaires attribuées fonctionnelles de la séquence.

Nous avons atteint notre objectif qui consistait à construire les différents filtres de couplage indépendamment les uns des autres. La seule fonction à créer quand la totalité de la séquence est connue, est la fonction Σ qui appelle correctement les différents filtres. Par contre la taille de l'évaluateur est plus grande que dans la construction présentée précédemment et le nombre de tests dans chaque filtre de couplage augmente.

4.4 Acceptation des dags dans les grammaires attribuées fonctionnelles

Pour la méta-composition les dags posent problème ; en effet l'algorithme génère sur les dags plusieurs définitions pour une même occurrence d'attribut (cf. figure 3.4).

L'idée vient donc de dupliquer dans la grammaire attribuée toutes les parties associées à un dag. Le problème de cette idée est qu'il est nécessaire de générer une grammaire attribuée correcte. Or, si un non-terminal auquel est attaché un attribut appartenant à un dag dérive dans lui-même, on ne sait que dynamiquement combien de fois l'attribut en question doit être dupliqué. De plus, si le problème est restreint aux non-terminaux qui ne dérivent pas dans eux-même, un nouveau problème est créé par la duplication de certains attributs. En effet, sur certaines productions la valeur des attributs dupliqués ne sera pas utilisée et donc la grammaire attribuée construira des forêts, ce qui pose des problèmes pour l'application de la méta-composition.

Dans la composition des évaluateurs, ces problèmes se résolvent facilement.

Il n'est pas nécessaire de déclarer l'ensemble des attributs statiquement et comme sur un arbre donné l'ensemble des attributs à dupliquer est borné, il faut trouver un mécanisme qui permet de déclarer dynamiquement de nouveaux attributs et qui permet d'y accéder facilement sans se tromper.

Comme la composition des évaluateurs permet de ne construire que ce qui est nécessaire, il n'y a plus non plus de problème de construction de forêt.

L'idée est d'associer à chaque branche d'un dag un unique identificateur et de rajouter à chaque occurrence d'attribut rencontrée en parcourant cette branche avec une séquence de visistes, l'identificateur de la branche. Le nom d'une occurrence d'attribut peut ainsi comporter plusieurs identificateurs de branches si, pendant la séquence de visite, plusieurs dags ont été rencontrés.

Tout se passe pour le mieux si l'on commence toujours par descendre sur un dag avant de remonter, c'est le cas si l'évaluateur paramétrable est un évaluateur qui démarre à la racine. Par contre, si c'est un évaluateur qui démarre sur n'importe quel nœud de l'arbre l'évaluateur risque de tomber sur un dag par en-dessous. Il ne sait pas alors de quelle branche il provient.

Nous allons donc uniquement présenter le cas simple d'un évaluateur Top-Down.

Modification de la construction

Pour chaque production, associer un identificateur à chaque apparitions dans une règle sémantique d'une même occurrence d'attribut. A une occurrence d'attribut et un identificateur est associée une unique position dans une règle sémantique. L'ensemble des identificateurs associés à une occurrence d'attribut est appelé ensemble d'annotations.

Définition 25 (Ensemble d'annotations) *Étant donnée une occurrence d'attribut $X_i.a$ d'une une production $p : X_0 \to X_1 \cdots X_n$, on appelle ensemble d'annotations associé à $X_i.a$, noté $\mathcal{N}_{X_i.a}$ un ensemble d'identificateurs ou annotations tel que :*

- *L'ensemble associé à une occurrence d'attribut qui n'apparaît que dans une seule règle sémantique est donc restreint à l'ensemble : $\{\epsilon\}$. Si par contre elle apparaît plusieurs fois $\{\epsilon\} \notin \mathcal{N}_{X_i.a}$.*

- *il existe un isomorphisme $\mu_{X_i.a}$, qui à une règle sémantique r et à une position dans cette règle sémantique k associe une unique annotation : $\delta = \mu_{X_i.a}(r, k)$. De plus, $\mu^{-1}_{X_i.a}(\delta) = (r, k)$.*

Nous supposons que toutes les annotations sont différentes.

Pour repérer quelle branche d'un dag l'évaluateur a suivi dans une grammaire attribuée donnée nous avons introduit la notion d'annotation de branche.

Définition 26 (Annotation de Branche) *Étant donnée une grammaire attribuée fonctionnelle AG, on appelle annotation de branche, notée Δ, une séquence d'annotation $\Delta = \delta^1.\cdots.\delta^m$[7] où chaque $\delta^k, k \in [1..m]$ est une annotation associée à une occurrence d'attribut sur une production de AG.*

Comme nous travaillons avec une séquence de grammaires attribuées, il est nécessaire d'introduire la notion d'annotation de séquence qui repère pour chaque grammaire attribuée de la séquence quelle branche l'évaluateur est en train de parcourir.

Définition 27 (Annotation de Séquence) *Étant donnée une séquence Σ_i^j, on appelle annotation de séquence un n-uplet $\Omega = (\Delta^i, \cdots, \Delta^j)$ tel que chaque $\Delta^k, k \in [i..j]$ est une annotation de branche pour la grammaire attribuée fonctionnelle AG^k*

Par facilité nous écrirons : $(\Omega, \Delta) = (\Delta^i, \cdots, \Delta^j, \Delta)$.

Afin de différencier les instances d'attributs d'une annotation (un parcours de branche particulier) de celles d'une autre annotation, nous avons introduit des instances d'attributs annotées.

Définition 28 (Instance d'Attribut Annotée) *On appelle instance d'attribut annotée une instance d'attribut à laquelle est associée à une annotation de séquence.*

Dans une instance virtuelle de production $V(p)$, à chaque nœud virtuel v est associé une unique annotation de séquence, on appelle une telle instance de production, instance virtuelle de production annotée. L'annotation permet de retrouver à quel parcours de branche appartient cette instance de production.

Définition 29 (Instance Virtuelle de Production Annotée) *Étant donnée une séquence Σ_1^j et une production $p : X_0 \rightarrow X_1 \cdots X_n$ de la grammaire G^j, une instance virtuelle de production annotée pour p est de la forme $VA(p) = (v_0, \Omega_0) \rightarrow (v_1, \Omega_1) \cdots (v_n, \Omega_n)$, avec :*

- *$v_0 \rightarrow v_1 \cdots v_n$ instance virtuelle de la production p pour Σ_1^j ;*
- *$\Omega_k, k \in [0..n]$ annotation de séquence pour Σ_1^j.*

L'annotation d'une instance de production est construite lors de la construction virtuelle.

[7] Comme précédemment $a.\epsilon = a$.

Définition 30 (Construction Virtuelle avec Annotation) *Soit $AG : G \rightarrow G'$ une grammaire attribuée fonctionnelle faisant partie d'une séquence Σ. On appelle construction virtuelle avec annotation sur une production $p : X_0 \rightarrow X_1 \cdots X_n$ suivant une occurrence d'attribut $X_j.a$ de cette production et une annotation de branche $\Delta = \delta_1.\cdots.\delta_m$, une fonction qui prend comme paramètres une instance virtuelle de production annotée pour p, la position j dans p du non-terminal qui porte l'occurrence d'attribut $X_j.a$ et renvoie une nouvelle instance de production annotée et une position dans celle-ci. On la note : $< VA(p), j > \underset{a,\Delta}{\Longrightarrow} < VA(p'), j' >$. Elle est définie de la façon suivante :*

1. *si $\mathcal{N}_{X_j.a} = \{\epsilon\}$ sur p alors $\delta = \epsilon$ sinon $\delta = \delta_1$ et $\Delta = \delta_2.\cdots\delta_m$;*
2. *soit $(f_{p,y^0,Y^0}, j') = \mu^{-1}_{X_i.a}(\delta)$ avec f_{p,y^0,Y^0} de la forme $Y^0.y^0 = p'(Y^1.y^1, \ldots, Y^l.y^l)$;*
3. *$VA(p') : (v'_0, \Omega'_0) \rightarrow (v'_1, \Omega'_1) \cdots (v'_l, \Omega'_l)$ avec pour tout $s \in \{0, \cdots, l\}$, si $Y^s = X_k$:*
   - *$v'_s = v_k.y^s$;*
   - *$\Omega'_s = (\Omega_k, \mu_{X_k.y^s}(f_{p,y^0,Y^0}, s).\Delta)$.*

Il faut interpréter cette construction comme suit :

- si l'évaluateur arrive sur une occurrence d'attribut qui est un dag alors l'évaluateur sort d'une branche "dupliquée". Il faut donc éliminer une annotation de l'indicateur de branche courant Δ. Ensuite, il faut construire pour chacun des nœuds virtuels l'indicateur de branche qui faudrait utiliser si l'évaluateur suivait ce nœud virtuel et l'ajouter à l'indicateur de séquence.
- si l'évaluateur arrive sur une occurrence d'attribut qui n'est pas un dag l'indicateur de branche courant ne change pas, mais il est quand même nécessaire de construire l'indicateur de branche pour chacun les nœuds virtuels, ceux-ci pouvant être des dags.

Si on construit une annotation de branche pour un nœud qui n'est pas un dag, ϵ est ajouté à l'annotation, donc l'annotation ne change pas.

Définition 31 (Filtre de Couplage avec Annotation) *Soit $AG : G \rightarrow G'$ une grammaire attribuée fonctionnelle, le filtre de couplage avec annotation de AG, noté FC, est un ensemble de transitions virtuelles avec annotation ayant la définition suivante :*

1. *pour chaque production $p : X_0 \rightarrow X_1 \cdots X_n$,*
2. *pour chaque règle sémantique $f_{p,y^0,Y^0} : Y^0.y^0 = p'(Y^1.y^1, \ldots, Y^l, y^l)$ sur p,*
3. *pour chaque occurrence d'attribut $Y^{j'}.y^{j'} = X_j.a, 0 \leq j' \leq l$ de cette règle sémantique ;*

on crée une transition virtuelle $< U, \Omega, VA(p), v(a), j > \Longrightarrow < U, \Omega', VA(p'), v(a'), j' >$ où :

U est le nœud père de l'instance de production courante de l'arbre réel ;

Ω *est une annotation de séquence.*

$V(p)$ *est une instance de production virtuelle pour une séquence de grammaires attribuées fonctionnelles se terminant par AG ;*

$v(a)$ *est un attribut d'une séquence de grammaires attribuées fonctionnelles commençant par AG ;*

j *est une position dans* p *;*

et telle que :

- $\Omega = (\Delta, \Omega')$;
- $v(a) = a.v(a')$;
- $< V(p), j > \underset{a,\Delta}{\Longrightarrow} < V(p'), j' >$.

Il reste maintenant à modifier un peu le sélecteur de visite et l'évaluateur paramétrable. Il suffit que ceux-ci acceptent les annotations :

- une nouvelle visite est appelée sur un nœud virtuel avec une annotation ;
- l'annotation passée au premier filtre de couplage est l'annotation du nœud virtuel d'appel ;
- l'évaluation et l'utilisation d'instances d'attributs se fait en utilisant les annotations.

L'annotation de départ est $(\epsilon, \cdots, \epsilon)$.

4.5 Optimisation de la fonction Σ pour diminuer les règles de copie

Nous allons essayer ici de diminuer les pertes en efficacité de l'évaluateur dues au parcours des règles de copie.

Nous savons que lorsqu'une instance de production de passage est obtenue en sortie d'un des filtres de couplage de la séquence, tous les autres qui le suivent vont générer des instances de productions de passage[8]. Il ne semble donc pas utile de parcourir le reste des filtres de couplage puisque grâce à l'attribut virtuel il est possible de déduire directement la production virtuelle finale. Une première optimisation de l'algorithme est donc possible.

Ensuite nous remarquons que s'il existe une suite de règles de copie partant d'un nœud virtuel v_0 et arrivant en v_n, seules les instances d'attributs portées par v_0 et v_n sont des instances d'attributs vraiment utiles, les autres ne servent qu'à propager l'information. L'idée est alors de ne faire la copie des instances d'attributs qu'entre v_0 et v_n. Pour cela,

[8]Par définition des productions de passage.

l'instance de production virtuelle $V(Id_{\mathcal{X}})\ : v_0 \rightarrow v_n$ est construite plutôt que chacune des instances de productions virtuelles intermédiaires ; la visite est alors appelée sur cette instance de production[9].

Pour faire ces optimisations, dans chacun des filtres de couplage, tous les cas correspondants à des productions de passage sont remplacés par un cas unique, qui ne fait que passer le quadruplet inchangé :

$$FC^i(< U, V(Id)\ : v_0 \rightarrow v_1, v(a), j >) =< U, V(Id)\ : v_0 \rightarrow v_1, v(a), j >$$

Une fonction d'optimisation, notée OP est ajoutée en sortie de Σ. Celle-ci devient alors $\Sigma = OP(FC^{m-1}(\cdots FC^1(SV(U,v))))$. La fonction OP est de la forme suivante :

$OP(< U, V(p)\ : v_0 \rightarrow v_1, v(a), j >)$
Si $p = Id_{\mathcal{X}}$ **faire**
  $< U', V(p')\ : v'_0 \rightarrow v'_1, v(a'), j' > \leftarrow < U, V(p)\ : v_0 \rightarrow v_1, v(a), j >$;
  Tant que $p' = Id_{\mathcal{X}'}$ **faire**
    $< U', V(p')\ : v'_0 \rightarrow v'_1, v(a'), j' > \leftarrow FC^{m-1}(\cdots FC^1(SV(U, v'_j.v(a'))))$;
  Fin tant que
  Si $j = 1$ **faire**
    Renvoie $(< U', V(p')\ : v'_0 \rightarrow v'_1, \epsilon, j' >, < U, V(Id_{\mathcal{X}})\ : v_0.v(a) \rightarrow v'_1.v(a'), \epsilon, j >)$;
  Sinon
    Renvoie$(< U', V(p')\ : v'_0 \rightarrow v'_1, \epsilon, j' >, < U, V(Id_{\mathcal{X}})\ : v'_0.v(a') \rightarrow v_1.v(a), \epsilon, j >)$;
  Fin si
Sinon
  Renvoie$(< U, V(p)\ : v_0 \rightarrow v_1, v(a), j >, < \epsilon, \epsilon, \epsilon, \epsilon >)$

Il faut également modifier l'évaluateur paramétrable afin que celui-ci accepte un couple en sortie de la fonction Σ.

Dans toutes les visites de l'évaluateur associées à des productions de passage l'appel à une nouvelle visite est éliminé et les visites deviennent :

`V`-num$(< U, V(p), \epsilon, j >, < U', V(Id), \epsilon, j' >)$
  Si $V(Id) \neq \epsilon$ **alors**
    Case Id :
      ⋮
      Case j' :
        ⋮
        `V`-num-Id-$j'(U', V(Id))$;
        ⋮

[9]Cette visite est la même que celle qui aurait été appelée sur chacune des instances de productions intermédiaires.

            Fin Case
        Fin Case
    Fin si
    Case $p \in P$:
        Case $j \in [0..n]$:
            $\vdots$
            V-num-p-$j(U, V(p))$;
            $\vdots$
            Fin Case
        Fin Case

Avec ces différentes modifications, l'évaluateur va suivre les règles de copie sans les construire complètement, tant qu'il ne trouve pas de construction. Quand il a trouvé une construction il copie effectivement les attributs entre les extrémités de la suite de règles de copie et appelle la visite sur la construction.

4.6 Comparaison avec d'autres travaux

Le seul résultat qu'il est possible de rapprocher de la composition des évaluateurs est la proposition donnée par Giegerich dans [39].

Proposition
Étant donnés deux évaluateurs pour $\alpha, \beta \in$ FNC/S ou l-ordonnée/1S[10] *et tels que $\beta \circ \alpha$ existe, il est possible de construire directement un évaluateur pour $\beta \circ \alpha$ de la même classe que ceux de β et α*

Par directement, l'auteur entend en fait qu'il n'est pas nécessaire de refaire la construction des relations de dépendances pour $\beta \circ \alpha$ car celles-ci se déduit directement des relations de dépendances de α et β. Il est donc toujours obligé de construire les séquences de visites de l'évaluateur.

C'est un peu la même idée qui est utilisée dans la composition des évaluateurs si ce n'est que l'on ne travaille plus au niveau des dépendances mais au niveau des évaluateurs.

4.7 Conclusion

J'ai présenté dans ce chapitre trois types de constructions pour créer un évaluateur pour une séquence de grammaires attribuée fonctionnelles.

Chacune s'approche un peu plus que la précédente d'une compilation séparée ; en contre partie elles perdent en efficacité des évaluations et ont une taille de plus en plus impor-

[10] Les classes FNC/S et $l - ordonne/1S$ sont respectivement la classe des grammaires attribuées FNC restreinte à l'ensemble des grammaires attribuées dont la partie syntaxique est purement synthétisée et la classe des grammaires attribuées l-ordonnée dont la partie syntaxique est purement synthétisée et ne porte que sur un unique attribut.

tante. Seule une implantation complète de ces trois constructions permettrait de quantifier correctement ces pertes.

Le net avantage de la composition des évaluateurs sur la méta-composition est qu'elle assure la construction d'un évaluateur par séquences de visites dans le cadre FNC. En contre-partie l'évaluateur généré est, dans certains cas, moins efficace que l'évaluateur qui aurait été obtenu après la méta-composition. La dernière construction a aussi l'avantage de permettre une compilation séparée totale.

Une voie de recherche qui me paraît intéressante est d'essayer d'optimiser encore les évaluateurs produits par la composition des évaluateurs en essayant de regrouper dans une unique séquence de visites plusieurs séquences de visites qui pourrait être faite en même temps. En effet, il existe des séquences de visites qui parcouraient deux branches différentes d'un arbre intermédiaire qui se retrouvent sur une même branche de l'arbre d'entrée, si celles-ci n'ont aucune interaction les unes avec les autres elle peuvent être regroupées. Cette propriété de non-interaction semble être la même que celle que l'on recherche dans le cadre de la parallélisation automatique des grammaires attribuées ; il semble donc intéressant de rapprocher ce travail de celui de Bruno Marmol [40].

Chapitre 5

Les Grammaires attribuées incomplètes

5.1 Présentation et définitions

Dans les chapitre précédents nous avons essayé de décrire différents outils permettant de développer une application de façon modulaire sans diminuer l'efficacité des évaluateurs produits. La modularité que nous avons décrit reposait sur la notion de grammaire attribuée fonctionnelle et sur la décomposition en phases.

Afin d'augmenter encore les possibilités de modularisation d'une spécification par grammaires attribuées, nous avons introduit la notion de grammaires attribuées incomplètes. Celles-ci permettent comme les grammaires attribuées fonctionnelles de décrire une grosse application sous forme de modules. La différence est qu'ici les modules ne sont pas *a priori* ordonnés en séquences mais peuvent très bien être interdépendants.

Les grammaires attribuées incomplètes favorisent plus le découpage en unité de calcul que les découpages en phases. Les différents modules communiquent entre eux grâce à des attributs d'entrée et de sortie. L'attribut de sortie contient une valeur qui est calculée dans la grammaire attribuée et l'attribut d'entrée contient une valeur qui a été calculée de façon externe.

Les grammaires attribuées incomplètes et les grammaires attribuées fonctionnelles introduisent donc deux notions complémentaires pour la modularité.

Cette nécessité de décomposition en unité de sens se retrouvedans dans d'autres travaux [5–7], notre approche est équivalente à celle de Farrow, mais les outils que nous proposons pour les exploiter sont très différents.

Dans ce chapiter, nous allons décrire de façon précise les grammaires attribuées incomplètes ainsi que l'ensemble des techniques qui permettrons de les tester.

Pour rendre les grammaires attribuées incomplètes plus indépendantes de la syntaxe de base, j'ai décidé de les décrire sur des grammaires où n'interviennent plus de terminaux. En effet, la notion de terminal n'a d'intérêt du point de vue grammaire attribuée (sémantique) que parce qu'il porte une valeur sémantique qui a été calculée de façon externe, or cette notion est incluse dans la notion d'attribut d'entrée des grammaires attribuées incomplètes et de plus dans une optique de réutilisabilité l'attribut d'entrée à l'avantage de pouvoir être aussi bien associé à la valeur d'un terminal qu'à une autre valeur. D'un point de vue théorique nous n'avons plus besoin de la notion de terminal.

J'introduis donc les grammaires purement syntaxiques.

Définition 32 (Grammaire Purement syntaxique) *On appelle grammaire purement syntaxique le 3-uplet* $(N, P, \mathcal{Z})$ *avec :*

- N *est un ensemble de non-terminaux ;*
- P *un ensemble productions nommées :*

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

  avec $type(X_i) \in N, 0 \leq i \leq n$ *;*

- $\mathcal{Z}$ *est la racine,* $\mathcal{Z} \in N$.

Deux productions nommées peuvent avoir le même profil sans avoir le même nom. Par contre, au prix d'un renommage, nous supposons que les noms de productions sont uniques. Il est clair qu'il n'est pas toujours possible de construire un analyseur syntaxique pour une grammaire purement syntaxique car elle est potentiellement ambiguë. Nous supposons donc que les arbres de cette grammaire sont déjà construits grâce à une grammaire indépendante du contexte. La transformation de l'arbre d'entrée en un arbre de la grammaire purement syntaxique peut se faire grâce à une grammaire fonctionnelle (cf. figure5.1).

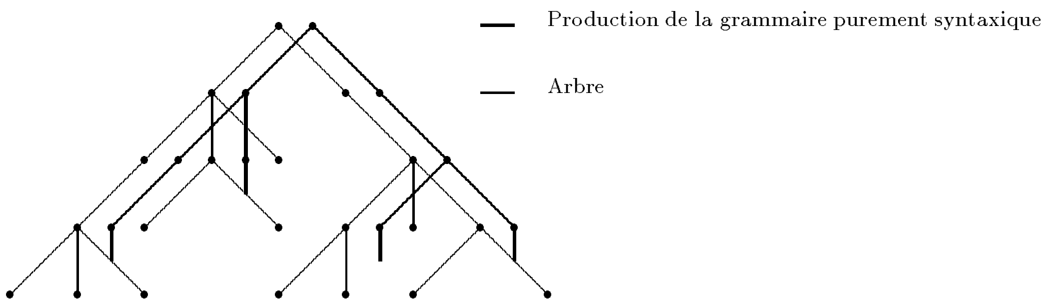


Figure 5.1: Exemple de passage d'un arbre réel vers un arbre “purement syntaxique”

Définition 33 (Grammaire Attribuée Incomplète) *Une* grammaire attribuée incomplète α *est composée des éléments suivants :*

- *une grammaire purement syntaxique* $G = (N, P, \mathcal{Z})$ *;*
- *une signature* $\Sigma = (\mathcal{K}, \mathcal{F})$*, avec* $\mathcal{K}$ *ensemble de types et* $\mathcal{F}$ *un ensemble de fonctions élémentaires de* $\mathcal{K}^*$ *dans* $\mathcal{K}$ *;*

- *un ensemble de types d'entrée* $\mathcal{I} \subset \mathcal{K}$ *;*
- *un ensemble de types de sortie* $\mathcal{O} \subset \mathcal{K}$ *;*
- *pour tout non-terminal* $\mathcal{X} \in N$ *un ensemble d'attributs* $\mathcal{A}(\mathcal{X}) = \mathcal{H}(\mathcal{X}) \cup \mathcal{S}(\mathcal{X})$. *Chaque attribut* a *est de type,* $type(a) \in \mathcal{K}$ *;*
- *pour toute production* $p : X_0 \rightarrow X_1 \cdots X_n$ *un ensemble d'attributs d'entrée* $I(p)$. *Un attribut d'entrée* $i(X_0)$ *est de type,* $type(i(X_0)) \in \mathcal{I}$.
- *pour toute production* $p : X_0 \rightarrow X_1 \cdots X_n$ *un ensemble d'attributs de sortie* $O(p)$. *Un attribut de sortie* $o(X_0)$ *est de type* $type(o(X_0)) \in \mathcal{O}$.

  Afin de faciliter les différentes définitions, sur une production $p : X_0 \rightarrow X_1 \cdots X_n$, *un attribut d'entrée ou de sortie* a *sera noté comme une occurrence d'attribut du non-terminal père de* p, $X_0.a$.

- *pour toute production* $p : X_0 \rightarrow X_1 \cdots X_n$ *un ensemble de règles sémantiques* E_p. *On définit :*

$$DO(p) = \bigcup_{1 \leq i \leq n} H(X_i) \cup S(X_0) \cup O(p)$$

l'ensemble des occurrences d'attributs définies dans p *;*

$$UO(p) = \bigcup_{1 \leq i \leq n} S(X_i) \cup H(X_0) \cup I(p)$$

l'ensemble des occurrences d'attributs utilisées dans p *;*

$$W(p) = UO(p) \cup DO(p)$$

l'ensemble des occurrences d'attributs de p.

Pour toute occurrence d'attribut, $X_k.a \in DO(p)$ *il existe une règle sémantique dans* E_p *de la forme :*

$$r_{p,a,k} : X_k.a = F_{p,a,k}(X_{f(1)}.a_1, \cdots, X_{f(q)}.a_q)$$

avec $X_{f(i)}.a_i \in UO(p)$, f *une fonction de* $\{1, \cdots, q\}$ *dans* $\{0, \cdots, n\}$ *et* $F_{p,a,k}$ *est l'abstraction d'un terme pour l'ensemble de variables* $\mathcal{V}$ *composé de l'ensemble des occurrence d'attributs utilisées dans* p.

Remarque Nous avons associé les attributs d'entrée et de sortie à une production afin de rendre notre définition la plus générale possible.

Évaluation d'une GAI

Nous allons présenter ici une méthode qui permet de tester une grammaire attribuée incomplète de façon "isolée". Pour cela il est nécessaire de trouver un moyen d'appeler une grammaire attribuée incomplète sur un arbre d'entrée et de calculer toutes les instances d'attributs de l'arbre.

Afin de rendre une grammaire attribuée incomplète évaluable sur un arbre réel il suffit que la grammaire couplée par attribut de passage laisse dans l'arbre de la grammaire purement syntaxique des instances d'attributs calculées de même nom que les attributs d'entrée.

Pour calculer les différentes instances d'attributs il faut construire un évaluateur pour la GAI.

La notion de non-circularité se généralise aux grammaires attribuées incomplètes. De même la caractérisation FNC, DNC ou l-ordonnée s'applique très bien pour les grammaires attribuées incomplètes.

Par contre pour les évaluateurs associés à ces classes, il faut effectuer quelques petites modifications afin de prendre en compte les valeurs des instances d'attributs d'entrée et afin d'être sûr que toutes les instances d'attributs de l'arbre ont bien été calculées.

Valeur des attributs d'entrée

Chaque fois que l'évaluateur a besoin de la valeur d'un attribut d'entrée dans une fonction d'évaluation, il fait appel à la fonction *get-value* qui prend comme paramètre une instance d'attribut d'entrée et qui renvoie une valeur à cette instance.

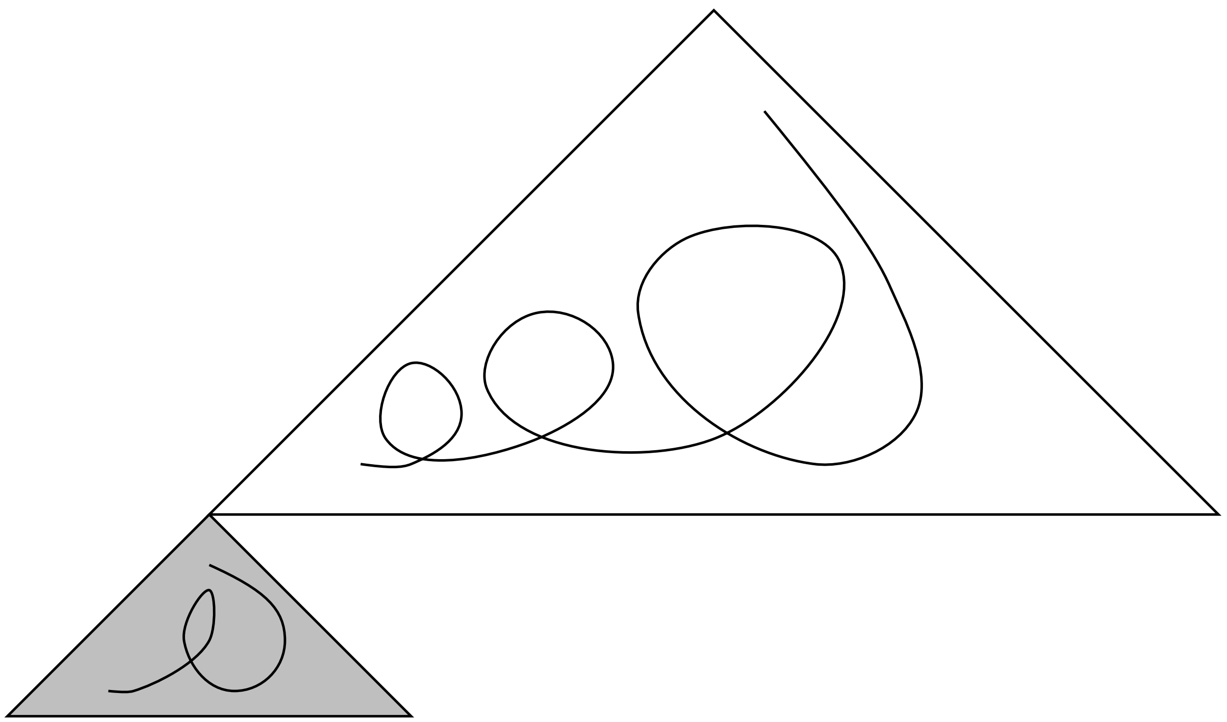
Calcul de toutes les instances d'attributs de l'arbre

Afin d'être sûr de calculer toutes les instances d'attributs de l'arbre, il est nécessaire de ne pas oublier de visiter certaines parties de l'arbre où il reste des instances d'attributs à calculer. Pour cela il est nécessaire, pour toute fonction d'évaluation lancée sur un u, de lancer :

- une visite sur chaque nœud ne portant pas d'instances d'attributs ;
- un visite sur un nœud u' différent de u après tout calcul d'une de ses instances d'attributs associées. Ceci implique que la dernière opération effectuée sur un nœud soit un appel à une visite sur ce nœud.

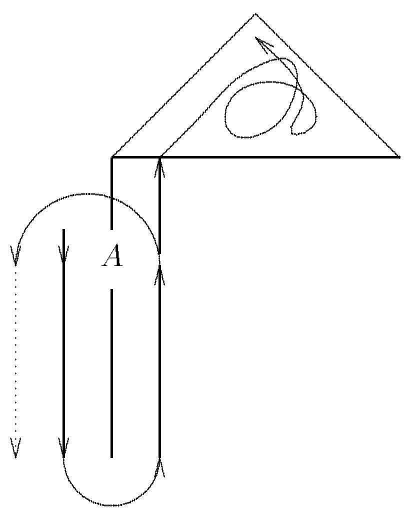
La première condition assure que les calculs qui sont isolés dans l'arbre ne sont pas oubliés (cf. Figure 5.2).

La deuxième condition assure qu'après tout calcul d'une instance d'attribut sur un nœud, l'évaluateur lance bien une visite dans laquelle les instances d'attributs qui utilisent l'instance qui vient d'être calculée, sont susceptibles d'être calculées (cf. Figure 5.3).



Dans un évaluateur FNC, classique, les calculs qui se trouvent dans la partie grisée ne seront jamais atteints.

Figure 5.2: Exemple



Dans un évaluateur FNC, classique, A n'est visité qu'une fois et les instances d'attributs sur les dépendances en pointillés ne seront jamais calculées.

Figure 5.3: Exemple

Construction d'une application complète

Nous avons toutes les techniques nécessaires afin de tester une grammaire attribuée incomplète de façon isolée. Nous allons maintenant nous intéresser à un ensemble de grammaires attribuées incomplètes definies sur une même grammaire purement syntaxique, en supposant que celles-ci décrivent une application complète.

5.2 Union des spécifications

Étant donné un ensemble de grammaires attribuées incomplètes définies sur une même grammaire purement syntaxique. Si l'on veut que les différents évaluateurs associés à ces grammaires attribuées incomplètes communiquent entre eux, il est nécessaire que lors de l'évaluation, les valeurs de certaines instances d'attributs de sortie d'une grammaire attribuée soient passées à l'évaluateur d'une autre grammaire attribuée incomplète. Pour plus de simplicité, nous supposons que cette transmission de valeur se fait entre attributs d'entrée et de sortie de même nom appartenant à une même production.

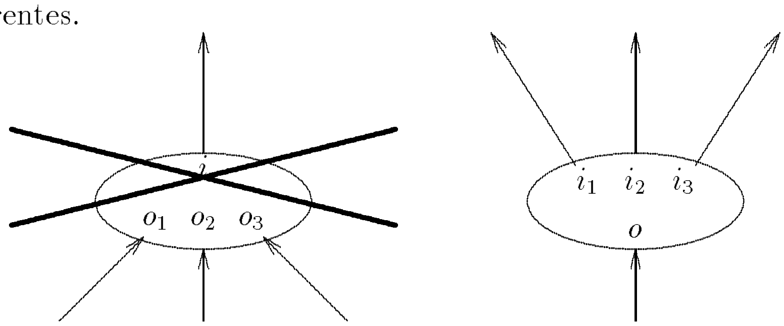
Ayant vérifié la validité de chacune des grammaires attribuées incomplètes prisent de façons séparées, il faut maintenant vérifier que leur association est valide.

Dire que cette association est valide signifie qu'il est possible de générer un évaluateur correct pour l'application.

La première condition est une condition locale, il ne faut pas qu'une occurrence d'attribut soit définie plusieurs fois dans une même production.

Les différentes grammaires attribuées incomplètes sont supposées correctes, prises indépendamment les unes des autres. Si nous supposons que les domaines des noms des attributs classiques sont, quitte à un renommage, distincts pour chaque grammaire attribuée incomplète, la première condition est vérifiée pour tous les attributs classiques.

Il reste donc à vérifier que pour chaque occurrence d'attribut d'entrée d'une production p, il existe une unique occurrence d'attribut de sortie de même nom sur p dans l'ensemble des grammaires attribuées incomplètes. Par contre, une occurrence d'attributs de sortie peut être associée à plusieurs attributs d'entrée provenant de grammaires attribuées incomplètes différentes.



Une deuxième condition pour que l'association soit valide est que l'association des différentes grammaires attribuées incomplètes n'entraîne pas la création de circularité. Nous savons que ce problème est exponentiel en la taille de la grammaire attribuée dans le cas classique. C'est pourquoi nous introduisons la notion de graphe d'entrée-sortie qui permet de déduire des propriétés intéressantes sur la circularité.

Pour cela nous utilisons la notion de dépendance entre attributs associés à un non-terminal, attributs d'entrée et attributs de sortie $\mapsto$.

Définition 34 (Relation de dépendance $\mapsto$ d'une grammaire attribuée incomplète) *La relation $\mapsto$ est définie sur l'ensemble des attributs d'une grammaire attribuée incomplètes, c'est-à-dire l'ensemble des attributs, des attributs d'entrée et des attributs de sortie. Il existe une relation entre deux attributs $b \mapsto a$, s'il existe, sur une production, une règle sémantique définissant une occurrence de a qui utilise une occurrence de b.*

Définition 35 (Relation d'Entrée-sortie d'une GAI) *On appelle* relation d'entrée-sortie *pour une grammaire attribuée incomplète, la relation sur l'ensemble des attributs d'entrée et de sortie de la grammaire attribuée incomplète, telle qu'il existe une relation $b \rightarrow a$ si et seulement si $b \mapsto^{+} a$.*

Étant donné l'ensemble des relations d'entrée-sortie associées aux différentes grammaires attribuées incomplètes composant une application, nous définissons la **relation d'entrée-sortie de l'application** comme l'union des différentes relations d'entrée-sortie, où les attributs d'entrée et attributs de sortie de même nom sont confondus.

Nous remarquons que la première condition pour que l'association soit valide se retrouve dans le graphe d'entrée sortie de l'application sous la forme :

il n'existe qu'un seul arc entrant sur chaque sommet.

La construction de la relation d'entrée-sortie de l'application nous permet dans la plupart des cas de conclure si l'application est circulaire, grâce à la propriété suivante :

Propriété 1 *Si chacune des grammaires attribuées incomplètes constituant une application est non-circulaire et que la relation d'entrée-sortie de l'application ne contient pas de circularité, alors l'association est valide.*

Cette condition est une condition suffisante mais pas nécessaire. Il est possible de trouver des associations valides telles que le graphe d'entrée-sortie soit circulaire (cf. figure 5.6)

Une fois que la validité d'une association a été vérifiée, il est possible d'adopter plusieurs optiques pour générer l'évaluateur de l'application, la méta-union ou la liaison des évaluateurs.

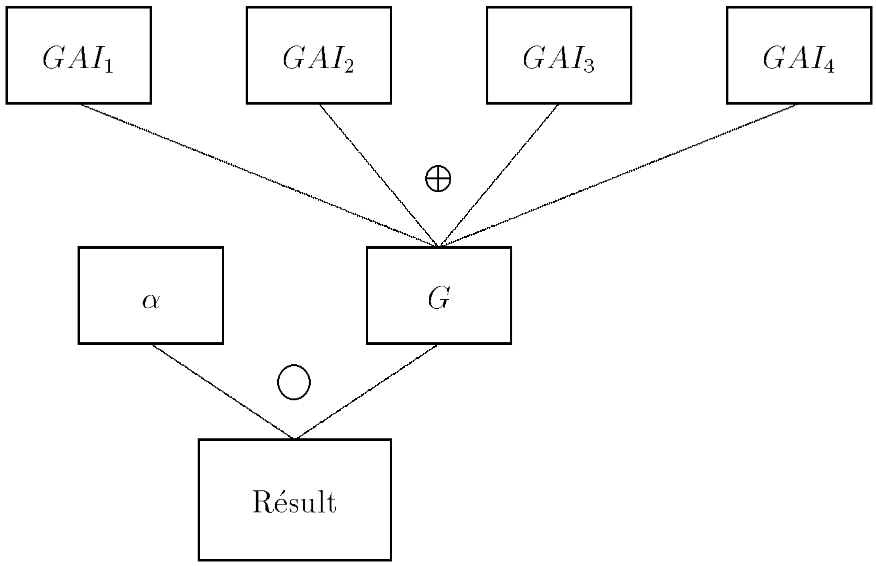
5.2.1 La Méta-Union

La méta-union consiste comme pour la méta-composition à générer à partir des différents modules décrivant l'application un unique grammaire attribuée G.

L'opération est assez simple, elle est composée des étapes suivantes :

- pour chaque attribut classique a de chaque grammaire attribuée incomplète α on déclare dans G l'attribut a_α ;
- pour chaque attribut de sortie de chaque grammaire attribuée incomplète, on génère un attribut local de même nom ;
- pour chaque attribut d'entrée de chaque grammaire attribuée incomplète, s'il n'existe pas dans G sur la même production un attribut local de même nom, on déclare un attribut d'entrée de même nom dans G ;
- pour chaque règle sémantique de chaque grammaire attribuée incomplète, on génère dans G une règle sémantique équivalente où chaque occurrence d'attribut a été remplacée par celle qui lui correspond dans G.

Nous nommons cet opérateur $\bigoplus$. Une fois que toutes les grammaires attribuées incomplètes de l'application sont regroupées dans un même grammaire incomplète G, il est intéressant de transformer celle-ci en un grammaire classique. Pour cela nous appliquons la méta-composition[1] sur la grammaire couplée par attributs qui fait passer d'un arbre d'analyse réel à un arbre de la grammaire purement syntaxique de base et sur la grammaire G.



Il est maintenant possible de générer un évaluateur pour la grammaire résultante.

La seule propriété que nous connaissions à ce point est que la grammaire attribuée résultante est non-circulaire. Comme nous désirons nous placer dans le cadre FNC, il nous faut connaître d'autres propriétés de la grammaire attribuée résultante. La propriété suivante nous permet dans la plupart des cas, de conclure si la grammaire attribuée résultante est FNC.

[1]Les extensions concernant les attributs exportés et importés nous permettent d'appliquer la méta-composition de façon "presque classique".

Propriété 2 *Si chacune des grammaires attribuées incomplètes constituant l'application est FNC et que le graphe d'entrée-sortie de l'application est non circulaire alors la grammaire résultante de la méta-union est FNC.*

Démonstration : Pour démontrer la propriété on introduit la définition suivante : *Soit une grammaire attribuée G, on appelle suite de construction d'une dépendance, la suite construite comme suit :*

Lors du test FNC (ou DNC), on se souvient récursivement à partir de quel suite de dépendances entre attributs $\mapsto$ *cette dépendance est construite.*

Pour chaque circularité trouvée, on construit en mettant bout à bout les suites de constructions associées aux dépendances constituant le cycle, une suite de dépendances entre attributs partant d'un attribut et revenant sur lui-même. On l'appelle suite de cycle.

Raisonnons par l'absurde. Supposons que l'on ait une circularité FNC (resp. DNC) dans la grammaire résultante, comme chaque partie la constituant est FNC (resp. DNC), la suite de cycle passe donc par des dépendances entre attributs provenant de différentes grammaires attribuée incomplètes. Il est possible d'extraire de la suite de cycle, des sous-suites constituées de dépendances n'appartenant qu'à une seule grammaire attribuée incomplète. Ces sous-suites relient un attribut $a.in$ et un attribut $b.out$ d'un même grammaire attribuée incomplète. Alors par définition il existe, dans le graphe d'entrée-sortie de la grammaire attribuée incomplète, un arc reliant $a.in$ et $b.out$. On obtient donc une circularité dans l'union des graphes d'entrée-sortie.

Il est sûr que cette propriété n'est intéressante que dans le cadre de la mise au point d'une application. Elle vérifie à moindre coût si, étape après étape, il est toujours possible de construire un évaluateur efficace pour la grammaire attribuée résultante de la méta-union. En effet, lors de la construction de l'évaluateur final, il nous faut faire la construction de la vraie relation FNC (resp. DNC). Par contre les différents tests FNC (resp. DNC) effectués sur les grammaires attribuées incomplètes prisent séparément sont utiles car l'union des relations FNC (resp. DNC) est incluse dans la relation FNC (resp. DNC) de la grammaire attribuée résultante de la méta-union, par définition de la méta-union.

Combinée avec la méta-composition, la méta-union nous permet dans la plupart des cas de construire un évaluateur efficace pour la grammaire attribuée résultante.

5.3 Liaison des évaluateurs

Une seconde approche comparable à celle choisie dans la composition des évaluateurs consiste à construire, à partir des différentes grammaires attribuées incomplètes et de leurs évaluateurs, un évaluateur pour l'application globale sans passer par la construction d'une grammaire attribuée. Nous ne travaillons pas au niveau de la spécification mais au niveau évaluateurs.

Nous allons décrire différents degrés de "raffinement" pour la construction de l'évaluateur de l'application globale.

Évaluateurs séquençables

Une première approche décrite dans [5] consiste à construire pour chacune des grammaires attribuées incomplètes un évaluateur classique. Nous cherchons ensuite à trouver statiquement un ordre valide d'appel aux différents évaluateurs précédemment construits. Si nous ne trouvons pas d'ordre valide, nous ne savons pas construire l'évaluateur global.

Par contre si nous trouvons un tel ordre le mécanisme de construction de l'évaluateur est très simple, celui-ci appelle les différents évaluateurs suivant l'ordre trouvé. L'évaluateur doit stocker dans l'arbre les instances d'attributs de sortie utiles aux évaluateur suivants.

Dans la figure 5.4 GAI1 fait des calculs sur l'arbre d'entrée et passe un certain nombre de valeurs à GAI2.

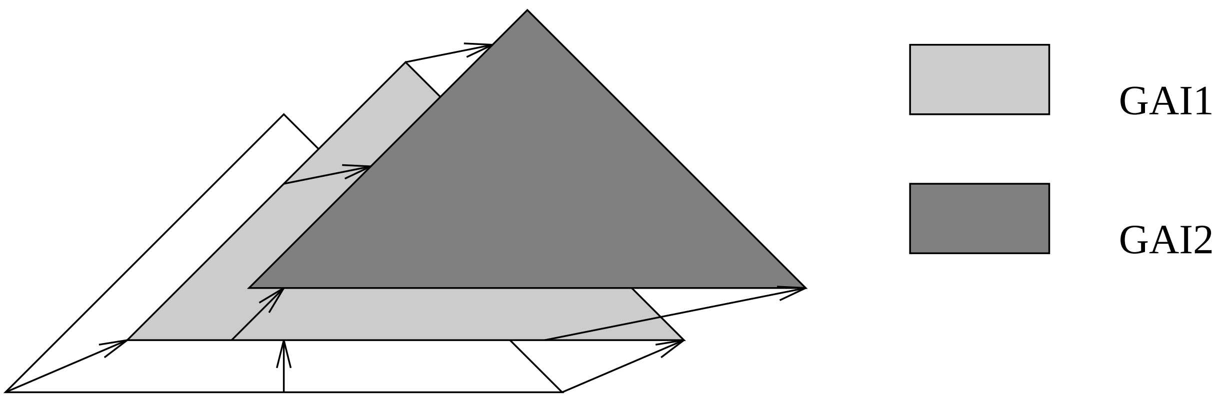


Figure 5.4: GAIs séquençables

Séparation Simple

Une amélioration de cette première solution consiste à décomposer chaque grammaire attribuée incomplète en un ensemble de sous-grammaires attribuées incomplètes dans lesquelles n'apparaissent que des attributs qui dans le graphe de la relation $\mapsto$ font partie d'une même partie connexe.

Nous déterminons les différentes parties connexes grâce à un algorithme classique. Nous construisons ensuite pour chaque partie connexe une grammaire attribuée incomplète en gardant la syntaxe de base et en éliminant toutes les déclarations et les règles sémantiques qui font intervenir des attributs qui n'appartiennent pas à la partie connexe.

Comme précédemment nous cherchons à ordonner statiquement ce nouvel ensemble de grammaires attribuées incomplètes. Le graphe d'entrée-sortie nous permet de trouver un ordre, s'il existe.

En effet, les parties connexes du graphe global se retrouvent dans le graphe d'entrée-sortie, car le graphe d'entrée sortie est basé sur la fermeture transitive de la relation $\mapsto$.

Dans la figure 5.5, la grammaire attribuée GAI1 est décomposée en deux sous-grammaires, la première appelle GAI2 et GAI2 appelle la seconde.

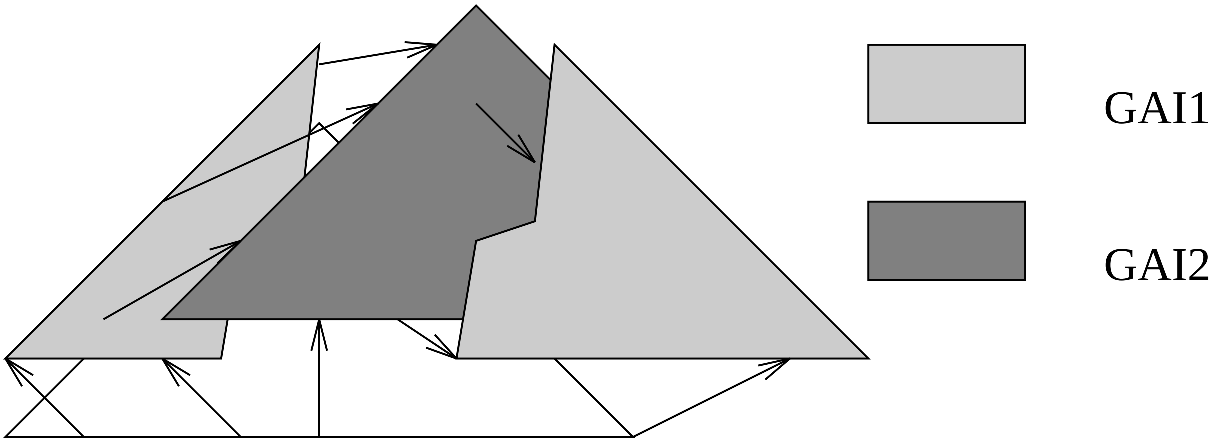


Figure 5.5: Sous-grammaires séquençables

Évaluateurs non séquençables

Nous nous intéressons maintenant à un ensemble de grammaires attribuées incomplètes pour lesquelles il n'est pas possible de construire un ordre statique. Celles-ci s'appellent mutuellement sur un même arbre. Nous prenons comme exemple les deux grammaires attribuées incomplètes GAI1 et GAI2, représentées dans la figure suivante 5.6. L'application peut se décomposer en un appel à GAI1, puis un appel à GAI2 et ainsi de suite.

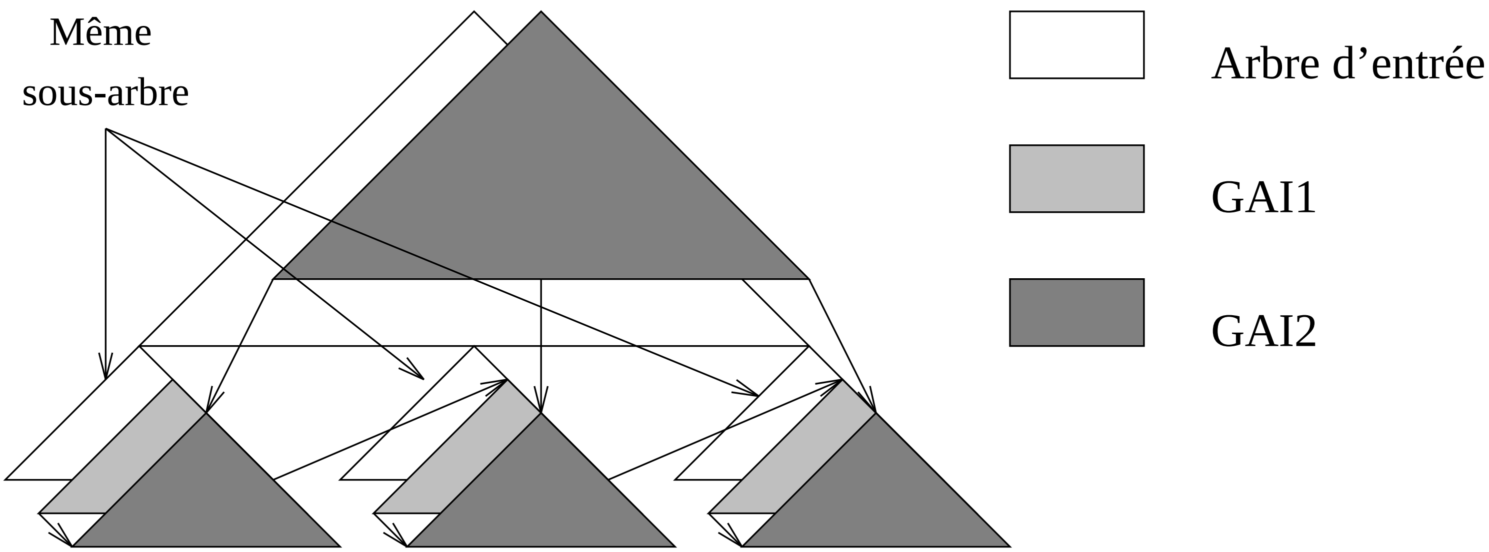


Figure 5.6: Exemple d'application décomposée en GAIs

L'idée est de ne plus construire l'ordre d'évaluation de façon statique mais de façon dynamique.

Chaque fois qu'un évaluateur a besoin de la valeur d'un attribut d'entrée, il appelle un autre évaluateur associé à l'attribut de sortie auquel est attaché l'attribut d'entrée. Celui-ci calcule l'attribut de sortie. L'évaluateur appelant peut alors récupérer la valeur de l'attribut d'entrée et reprendre son évaluation.

Par souci d'efficacité, nous allons essayer de construire un évaluateur qui calcule le nombre le plus faible possible d'instances d'attributs sachant qu'il doit renvoyer la valeur de l'attribut de sortie. Nous voulons également qu'il effectue le nombre le plus faible possible de visites dans l'arbre. C'est pourquoi nous utilisons un évaluateur basé sur une méthode statique.

Nous savons d'après [28] que lorsqu'une grammaire est DNC, nous sommes capables de construire un évaluateur basé sur une méthode statique qui peut être lancé à partir de n'importe quelle instance de production de l'arbre. Nous utilisons ce type d'évaluateurs. Ceux-ci peuvent être des évaluateurs DNC ou être issus de la transformation DNC vers l-ordonnée. Nous allons traiter plus complètement le cas de la transformation car c'est l'approche prise au sein du système FNC-2. Certaines décompositions et certains tests présentés plus loin ne sont pas nécessaires dans le cas d'un évaluateur purement DNC.

D'autre part, pour calculer le nombre minimal d'instances d'attributs, l'évaluateur utilisé n'est pas exhaustif contrairement à celui utilisé pour tester les différentes grammaires attribuées incomplètes.

Construction séparée des évaluateurs Pour chaque attribut de sortie nous cherchons à construire un évaluateur capable de démarrer son évaluation sur la production qui porte l'attribut de sortie. Nous nous assurons que cet attribut de sortie sera bien calculé à la fin de l'évaluation.

Comme nous cherchons à calculer un nombre minimal d'instances d'attributs de l'arbre, nous restreignons la grammaire attribuée incomplète. Pour un attribut de sortie a, nous déterminons tous les attributs qui dépendent de a au sens $\mapsto^{-1}$. Nous construisons ainsi l'ensemble minimal des attributs nécessaires au calcul de a quel que soit l'arbre d'entrée (ce minimum est un minimum relatif à la relation $\mapsto$ qui est une approximation de la relation dépendance réelle sur un arbre).

Nous restreignons alors la grammaire attribuée incomplète à cet ensemble d'attributs et aux règles sémantiques qui les définissent.

Ensuite, nous construisons un évaluateur pour cet nouvelle grammaire attribuée incomplète capable de commencer son évaluation sur la production portant l'attribut de sortie.

Liaison des évaluateurs Une fois ce travail effectué un évaluateur est associé à chaque attribut de sortie. Il faut maintenant construire l'évaluateur pour la totalité de l'application.

Pour lier les évaluateurs ainsi produits, nous utilisons la fonction *get-value* présentée précédemment. Chaque fois que l'évaluateur a besoin de la valeur d'un attribut d'entrée, la fonction *get-value* est appelée.

Cette fonction se comporte de la façon suivante lorsqu'elle est appelée pour une instance d'attribut de sortie a appartenant à une instance de production p :

- S'il y a un attribut de sortie qui a pour nom a sur l'instance de production p :
  - Si l'attribut de sortie a a déjà une valeur, l'attribut d'entrée prend cette valeur ;
  - Si l'attribut de sortie a n'a pas de valeur, on lance l'évaluateur auquel a est associé sur l'instance de production p ; une fois cette évaluation terminée l'attribut d'entrée a prend la valeur de l'attribut de sortie a ;
- Sinon il y a une erreur.

Contrôle de circularité Dans cette construction il n'y a aucun contrôle au niveau de la circularité, si ce n'est au niveau de chaque sous-grammaire au moment du test DNC. Ce test nous assure que chaque sous-grammaire incomplète est non-circulaire mais rien n'assure que l'évaluateur pour l'application n'est pas "circulaire".

Nous ajoutons donc un test de non-circularité, qui nous dira de façon dynamique si l'évaluateur global est circulaire ou non.

La fonction *get-value* vérifie avant de lancer tout évaluateur sur une instance de production s'il n'a pas déjà été lancé une fois. Comme les évaluateurs associés aux différentes sous-grammaires attribuées sont non-circulaires, s'il y a une circularité dans l'évaluateur global, elle se détectera sur la jonction entre un attribut d'entrée et un de sortie. De même pour les circularités dans les dépendances.

Malgré la décomposition de chacune des GAIs il est possible qu'il existe des circularités dans les évaluateurs qui ne correspondent pas à des circularités réelles. Prenons un exemple (cf. figure 5.7) :

Supposons que les règles sémantiques et les attributs nécessaires au calcul d'un attribut de

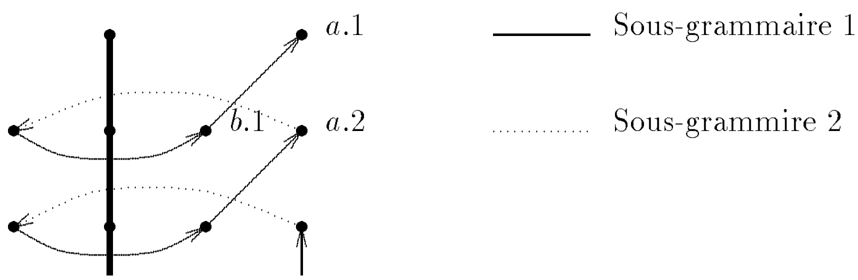


Figure 5.7:

sortie a ont été isolés dans GAI. Pour calculer l'instance a.2 de l'attribut de sortie a, l'évaluateur demande la valeur d'une instance d'attribut d'entrée b.1 à cause de l'approximation *du sélecteur d'arguments. En effet, b.1 n'est pas reliée à a.2 dans l'arbre (elle participe au calcul d'une autre instance de a, a.1 c'est pourquoi elle a été gardée dans GAI). Comme le calcul de la valeur de b.1 nécessite la valeur de a.2 et que celle-ci n'a a priori pas encore été calculée, l'évaluateur associé à a.2 est relancé. Ceci crée une circularité dans l'évaluation bien qu'il n'existe pas de circularité réelle*

Remarque : Dans le cas des évaluateurs séquençables, l'enchaînement des différents évaluateurs peut très bien se faire grâce la fonction *get-value* qui vérifie uniquement que l'évaluateur global n'a pas déjà été appelé précédemment.

Séparation fine

Dans l'exemple proposé de la figure 5.6, il y a une partie du calcul de GAI2 qui est commune aux différentes instances d'attributs de sortie. Avec la méthode présentée précédemment les évaluateurs construits recalculent cet ensemble d'instances d'attributs.

Pour chaque attribut de sortie qui les utilisent ces parties de calcul, l'évaluateur associé contient un même sous-ensemble de fonctions d'évaluation correspondant aux parties de calculs communes aux calculs des attributs de sortie.

Pour éviter ces recalculs, on peut utiliser une variante d'évaluateur de type incrémental. Celui-ci vérifie avant de lancer une séquence de visites, si certaines instances d'attributs ont déjà étaient calculées. Dans le cas incrémental il vérifie si ces instances d'attributs ont changé de valeurs.

Le problème d'un tel évaluateur est qu'il est nécessaire de stocker tous les attributs dans l'arbre. Il n'est pas alors possible de faire des optimisations mémoire dans l'évaluateur. De plus, il conserve l'inconvénient de construire des séquences de visites dupliquées.

En fait, nous remarquons que ce test n'a pas besoin d'être fait avant tout appel à une nouvelle visite. Certaines séquences de visites sont toujours nécessaires à partir du moment où la première visite est nécessaire.

Afin d'éviter tout test pendant l'appel à l'évaluateur nous avons à nouveau décomposé chacune des sous-grammaires en plusieurs grammaires incomplètes. Cette décomposition nous assure que l'appel à chacun des évaluateurs entraîne un minimum de recalcul et que les fonctions d'évaluation ne sont pas redondantes.

Séparation

Pour chaque attribut de sortie a, nous déterminons l'ensemble maximal des attributs qui ne dépendent que de cet attribut de sortie au sens $\mapsto^{-1}$, noté D_a.

Pour chacun des attributs de sortie nous construisons une grammaire attribuée incomplète ne contenant que les attributs de D_a et les règles sémantiques de la grammaire attribuée de départ les définissant. Pour chaque occurrence d'attribut $X.a$ qui apparaît en partie gauche d'une règle sémantique et qui ne fait pas partie de l'ensemble des attributs de D_a, un attribut d'entrée est déclaré. Il a le même nom a et est associé à toutes les productions où $\mathcal{X}$ apparaît en partie gauche.

Tous les attributs des différents D_a et toutes les règles sémantiques les définissant de la grammaire attribuée incomplète de départ sont éliminés. Pour chaque attribut d'entrée créé à l'étape précédente, un attributs de sortie correspondant est créé.

L'opération est appliquée de nouveau tant qu'il reste des attributs.

Afin d'éviter de trop morceler la grammaire attribuée de départ, il est intéressant de regrouper les différents attributs de sortie provenant d'un même attribut a dans un même évaluateur. Nous sommes sûrs de ne pas générer de circularité dans l'évaluateur en regroupant ces attributs puisqu'ils faisaient déjà partie d'une même grammaire attribuée non-circulaire.

Dans les grammaires attribuées incomplètes obtenues nous sommes sûrs, à l'approximation de la relation $\mapsto$ près, que tous les attributs servent uniquement au calcul de l'attribut de sortie a.

Pour chacune de ces nouvelles grammaires attribuées on construit un évaluateur basé sur la classe DNC qui est capable de partir de chacune des instances d'attributs de sortie. On lie les différents évaluateurs comme précédemment.

A moins que l'évaluateur associé à une instance d'attribut de sortie n'appelle des calculs associés à une autre instance du même attribut (cf. Figure 5.7), il n'y a pas de recalcul des instances d'attributs.

Remarques L'évaluateur global ainsi généré a plusieurs avantages :

- il coupe certaines circularités FNC ;
- il permet une compilation séparée des différentes grammaires attribuées incomplètes.

Il a par contre les inconvénients suivants :

- Certaines circularités de l'évaluateur global peuvent être évitées avec la méta-union ;
- Certaines instances d'attributs sont recalculées ;
- Au pire on construit un évaluateur ERN [22] pour l'évaluateur global.

5.3.1 Décomposition de la grammaire attribuée incomplète

La classe des grammaires attribuées DNC est plus petite que la classe des grammaires attribuées qui nous intéresse pour l'évaluation des grammaires attribuées incomplètes. En effet, la classe DNC implique que l'évaluation puisse débuter à partir de n'importe quelle instance de production de l'arbre. Dans le cas des grammaires attribuées incomplètes l'évaluation ne peut débuter que sur certaines productions ou certains non-terminaux bien précis. Il paraît donc intéressant de trouver une nouvelle classe de grammaire attribuée mieux adaptée à notre problème que la classe DNC.

Pour cela nous allons utiliser le test FNC, la composition des évaluateurs et une décomposition de la grammaire attribuée incomplète.

L'idée est la suivante :

- nous décomposons la grammaire attribuée incomplète en deux grammaires attribuées dont l'une porte l'attribut de sortie sur sa production racine ;
- nous appliquons le test FNC sur cette dernière grammaire attribuée et nous construisons l'évaluateur associé ;
- nous appliquons la composition des évaluateurs d'après la décomposition de départ.

Nous obtenons ainsi un évaluateur pour la grammaire attribuée de départ capable de débuter son évaluation sur la production portant l'attribut de sortie ; nous verrons d'après la construction que la classe des grammaires qui acceptent une telle construction est plus large que la classe DNC.

Il est possible d'élargir cette construction à plusieurs attributs de sortie en construisant une production racine pour chaque attribut de sortie.

Décomposition simple

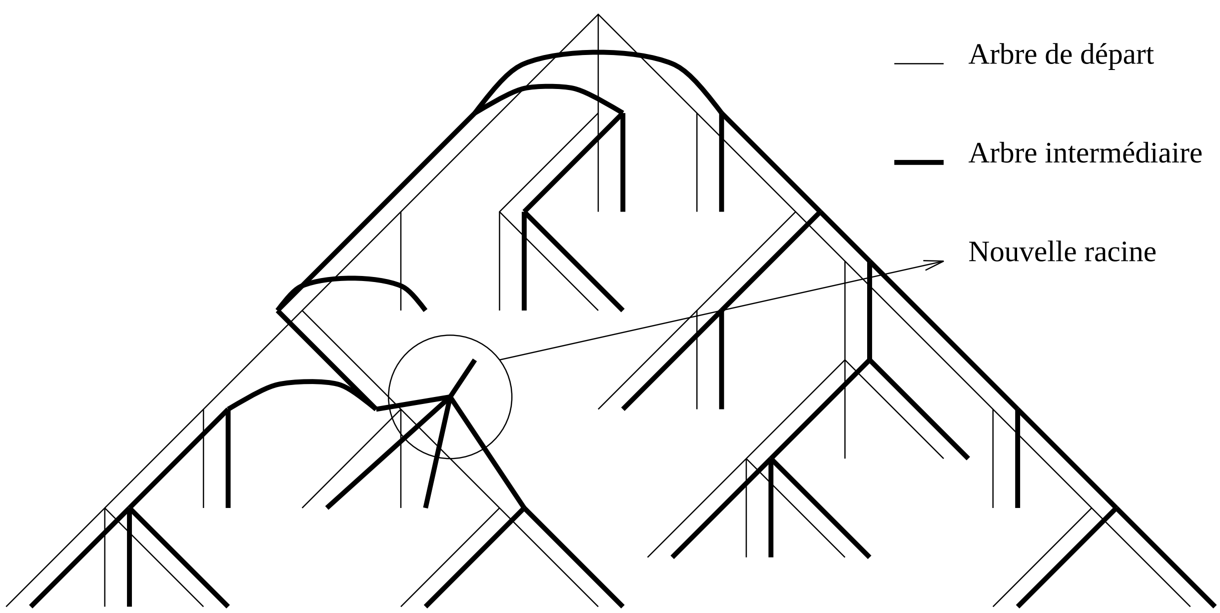


Figure 5.8: Exemple d'arbre intermédiaire pour la décomposition

Un première décomposition possible consiste à construire une grammaire attribuée fonctionnelle qui, sur un arbre donné, parcours toutes les branches de l'arbre une fois et une seule en partant de la production qui porte l'attribut de sortie. Un exemple est donné dans la figure 5.8. Ensuite, on fait en sorte de projeter convenablement les règles sémantiques de la grammaire attribuée incomplète de départ sur les constructions nécessaires à la construction de la grammaire attribuée fonctionnelle. La projection est convenable si la méta-composition des deux grammaires attribuées ainsi générées est bien égale à la grammaire attribuée de départ.

Soit γ la grammaire que nous cherchons à décomposer en $\alpha : G_1 \rightarrow G_2$ suivi de $\beta : G_2 \rightarrow G_3$. Nous appelons $G_2 = (N_2, T_2, P_2, \mathcal{Z}_2)$ grammaire intermédiaire.

L'algorithme de décomposition s'effectue alors comme suit :

1. on génère une racine $\mathcal{Z}_2$ pour la grammaire intermédiaire ;
2. on déclare l'attribut de sortie sur $\mathcal{Z}_2$ dans β ;
3. dans α, sur la production p qui porte l'attribut de sortie on déclare un attribut de type $\mathcal{Z}_2$, noté z_2 ;
4. dans α, sur p, on déclare un attribut hérité sur le non-terminal père de p et un attribut synthétisé sur chaque fils de p, leur type est le type du non-terminal qui les porte ;
5. dans α, sur p, on génère une construction définissant z_2 et dont les fils sont les attributs déclarés précédemment ;
6. dans G_2 on ajoute cette nouvelle construction ;
7. on transpose les attributs et les règles sémantiques de γ qui définissent l'attribut de sortie, sur cette nouvelle construction ;
8. tant qu'il existe une occurrence d'attributs $X_i.a$ sur une production $p : X_0 \rightarrow X_1 \cdots X_n$, non définie dans α, on cherche à la définir ;
9. on déclare, si besoin est, un attributs b sur chaque occurrence de non-terminal X_j de p différente de X_i, comme pour la production portant l'attribut de sortie ;
10. on ajoute dans α une construction définissant $X_i.a$ est dont les fils sont les $X_j.b$;
11. on transpose dans β les attributs et les règles sémantiques portées par p sur cette nouvelle construction.

Nous avons donné ici une idée de l'algorithme, nous en donnons maintenant une description formelle. Pour cela nous définissons une première fonction *Créer-construction* qui, étant donnée une occurrence d'attribut $X_i.a$ sur une production p, non définie dans β, va créer sa définition dans α, mais va également générer la sémantique sur cette nouvelle construction pour β.

Les tests entre parenthèses ne servent que sur la production qui porte l'attribut de sortie. Pour comprendre l'algorithme, on peut commencer par les oublier.

Fonction Créer-construction $(X_k.a, p)$

$A \leftarrow \emptyset$;
Pour chaque X_i **dans** $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
  Si $i \neq k$ (**ou** $a = z$) **alors**
    Si $i = 0$ **alors**
      Si $\mathcal{X}'_0 \notin N_2$ **alors**
        $N_2 \leftarrow N_2 \cup \{\mathcal{X}'_0\}$;
        $\mathcal{S}_\beta(\mathcal{X}'_0) \leftarrow \mathcal{H}_\gamma(\mathcal{X}_0)$;
        $\mathcal{H}_\beta(\mathcal{X}'_0) \leftarrow \mathcal{S}_\gamma(\mathcal{X}_0)$;
      Fin si
      Si $\mathcal{H}_\alpha(\mathcal{X}_0) = \emptyset$ **alors**
        $\mathcal{H}_\alpha(\mathcal{X}_0) \leftarrow \{c\}$ **avec** $type(c) = \mathcal{X}'_0$;
      Fin si
    Sinon
      Si $\mathcal{X}_i \notin N_2$ **alors**
        $N_2 \leftarrow N_2 \cup \{\mathcal{X}_i\}$;
        $\mathcal{S}_\beta(\mathcal{X}_i) \leftarrow \mathcal{S}_\gamma(\mathcal{X}_i)$;
        $\mathcal{H}_\beta(\mathcal{X}_i) \leftarrow \mathcal{H}_\gamma(\mathcal{X}_i)$;
      Fin si
      Si $\mathcal{S}_\alpha(\mathcal{X}_i) = \emptyset$ (**ou** $\{z\}$) **alors**
        $\mathcal{S}_\alpha(\mathcal{X}_i) \leftarrow \{c\}$ **avec** $type(c) = \mathcal{X}_i$;
      Fin si
    Fin si
    $A \leftarrow A \cup \{X_i.c\}$;
  Fin si
Fin pour
Soit $A = \{X_1.c_1, \cdots, X_n.c_n\}$; /* A est éventuellement vide */
  $P_2 \leftarrow P_2 \cup \{p' : type(X_k.a) \rightarrow type(X_1.c_1) \cdots type(X_n.c_n)\}$;
Pour chaque $r_{p,f(0),b_0} : X_{f(0)}.b_0 = F(X_{f(1)}.b_1, \cdots, X_{f(m)}.b_m)$ **dans** γ **faire**
  Soit $X'_{f'(i)}.c_i$ **l'équivalent dans** β **de** $X_{f(i)}.b_i$ **pour** $i \in [0..m]$;
  Ajouter $r_{p',f'(0),c_0} : X'_{f'(0)}.c_0 = F(X'_{f'(1)}.c_1, \cdots, X'_{f'(m)}.c_m)$ **à** β ;
Fin pour

Fonction Décomposition

$N_2 \leftarrow \{\mathcal{Z}_2\}$;
Soit $p : X_0 \rightarrow X_1 \cdots X_n$ la production qui porte l'attribut de sortie ;
  $\mathcal{S}_\alpha(\mathcal{X}_0) \leftarrow \{z\}$ **avec** $type(z) = \mathcal{Z}_2$;
  Associer dans β **l'attribut de sortie à** $\mathcal{Z}_2$;
  Créer-construction$(X_0.z, p)$;

Répéter
  Cond $\leftarrow$ *True*
  Pour chaque $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
    Si $\exists X_k.a$ occurrence d'attribut de α non définie **alors**
      Cond $\leftarrow$ *false* ;
      Créer-construction($X_k.a, p$) ;
    Fin si
  Fin pour
Tant que Cond ;

- On construit alors un évaluateur pour β de type FNC ;
- On applique la composition des évaluateurs pour obtenir l'évaluateur de l'application globale.

La complexité de la construction est linéaire en la taille de γ puisque la taille de la grammaire G_2 est linéaire en la taille de G_1. Pour chaque non-terminal de G_1 il existe au plus deux non-terminaux dans G_2. Pour chaque production $p : X_0 \rightarrow X_1 \cdots X_n$ de G_1 il existe au plus $n+2$ productions dans G_2.

Le test pour construire l'évaluateur de β est moins contraignant que le test DNC complet. En effet, on ne retrouve les "vraies" dépendances DNC que sur la partie de la grammaire qui correspond à la branche qui remonte dans l'arbre (*a priori* cela ne correspond pas à toute la grammaire) ; sur le reste de la grammaire on obtient uniquement les dépendances FNC. L'évaluateur généré est plus petit que l'évaluateur DNC puisqu'il ne peut débuter son évaluation que sur une production.

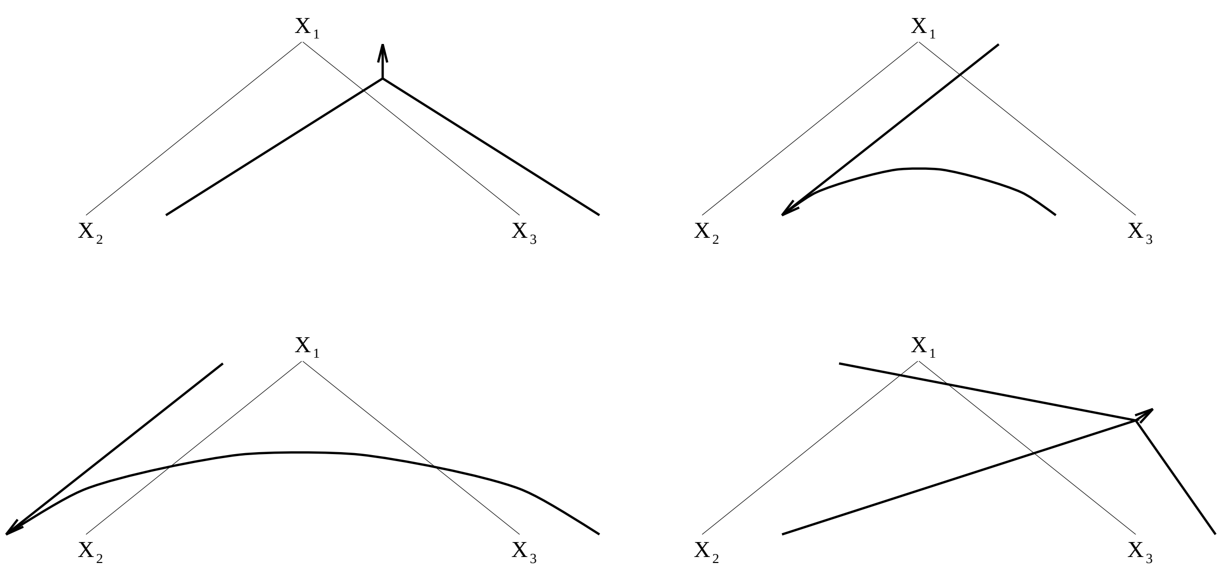


Figure 5.9: Constructions sur une même production pour le test DNC

Si nous désirons obtenir un évaluateur DNC complet avec une telle méthode il faut être capable de débuter l'évaluation sur n'importe quelle production. Ceci signifie qu'il est nécessaire de faire une construction racine sur toutes les productions. D'autres part, il est possible de remonter sur toutes productions par n'importe lequel des fils (contrairement à l'exemple précédent où l'évaluation remontait uniquement sur certains fils). Donc pour construire α et β, si nous désirons obtenir un évaluateur DNC, il faut :

- associer un attribut hérité et un attribut synthétisé à chaque non-terminal ;
- associer un attribut local à chaque production ;
- associer à chaque production $p : X_0 \rightarrow X_1 \cdots X_n$ $n+2$ règles sémantiques qui sont des constructions, représentées dans la figure 5.9.
- adapter les règles sémantiques associé à chaque production pour les associer aux constructions produites précédemment.

Conclusion

Seule une implantation de cet algorithme pourrait nous permettre de savoir si dans les cas pratiques cette augmentation de classe est intéressante.

Cette décomposition montre que la composition des évaluateurs peut permettre de construire des classes d'évaluateurs plus adaptées aux problèmes rencontrés dans les grammaires attribuées.

Chapitre 6

De nouvelles classes de grammaires attribuées

6.1 Décomposition pour l'évaluateur des GAIs

6.2 Les nouvelles classes de grammaires attribuées

Les chapitres précédents nous ont permis d'avoir une vision différente des évaluateurs de grammaires attribuées. Les évaluateurs ne parcourent plus uniquement les arbres de haut en bas ou de bas en haut mais peuvent parcourir une même branche dans plusieurs sens. C'est cette vision différente des évaluateurs qui nous a amenée à proposer une nouvelle classe DNC, un peu plus large que celle décrite classiquement 2 qui garde quand même les bonnes propriétés du test DNC.

Je vais donc introduire ici un nouvel algorithme pour la construction de la famille de relations R_{DNC} ainsi qu'un nouveau test d'appartenance à la classe DNC qui permettent d'étendre la classe DNC. Cette nouvelle construction et ce nouveau test sont les pendants naturels de la construction de la famille de relations R_{FNC} et du test FNC. Pour cela il faut considérer l'évaluateur DNC comme un évaluateur FNC partant d'un nœud quelconque de l'arbre et travaillant sur le sous-arbre, auquel a été ajouté un autre évaluateur FNC qui lui travaille sur le sur-arbre du nœud qui aurait été retourné.

Je vais introduire également de nouvelles classes de grammaires attribuées FNC+ et DNC+ qui sont des extensions naturelles des classes FNC et DNC.

6.3 Le nouveau test DNC

L'introduction de ce nouvel algorithme et de ce nouveau test provient de la constatation que dans le test classique DNC, le fait que sur une instance de production d'un arbre donné le parcourt se fait toujours dans le même sens n'est pas pris en compte. C'est-à-dire, soit l'évaluateur provient du nœud père et alors il descend vers les fils, soit il provient d'un fils et alors il remonte sur le père et descend sur les autres fils. Il ne faut donc pas mélanger l'ordre induit par en-dessous (test FNC) avec l'ordre induit par en-dessus (test DNC). La nouvelle famille d'ordre partiel $\mathcal{R}_{DNC'}(\mathcal{X})_{\mathcal{X}\in N}$ est donc telle que $R_{DNC'}(\mathcal{X}) \subseteq \mathcal{S}(\mathcal{X}) \times \mathcal{H}(\mathcal{X})$. La relation dérivée $R_{DNC'}(X_i)$ pour une occurrence de non-terminal est définie de façon classique. Par contre, le graphe de dépendances augmenté sur une production et sa projection sur une occurrence de non-terminal sont définis de façon particulière. Ils sont définis comme suit :

- $R_{DNC'}(p) = D(p) \cup R_{DNC'}(X_0) \cup \left(\bigcup_{1\leq i\leq n} R_{FNC}(X_i)\right)$ sur $W(p)$;
- $\mathcal{R}_{DNC'}(p, \mathcal{X}_i) = \left[D(p) \cup R_{DNC'}(X_0) \cup \left(\bigcup_{\substack{1\leq j\leq n \\ j\neq i}} R_{FNC}(X_j)\right)\right]^+ / \mathcal{X}_i$, la projection sur $\mathcal{A}(\mathcal{X}_i)$ de la restriction de $\left[D(p) \cup R_{DNC'}(X_0) \cup \left(\bigcup_{\substack{1\leq j\leq n \\ j\neq i}} R_{FNC}(X_j)\right)\right]^+$ aux éléments de $A(X_i)$.

Il est maintenant possible de définir la classe des grammaires attribuées DNC$'$.

Définition 36 (Classe DNC$'$) *Une grammaire est DNC' si et seulement si elle est FNC et s'il existe une famille de relations d'ordre partiel $\mathcal{R}_{DNC'}(\mathcal{X})_{\mathcal{X}\in N}$ telle que pour toute production de la forme $p : X_0 \to X_1 \cdots X_n$ on ait les propriétés suivantes :*

- **clôture :** $\forall i, 1 \leq i \leq n, \mathcal{R}_{DNC'}(p, \mathcal{X}_i) \subseteq \mathcal{R}_{DNC'}(\mathcal{X}_i)$;
- **non-circularité locale :** $\forall i, 1 \leq i \leq n, R_{DNC'}(p, X_i)$ *sans circuit ;*
- **non-circularité de départ :** $\mathcal{R}_{DNC'}(\mathcal{X}) \cup \mathcal{R}_{FNC}(\mathcal{X})$ *sans circuit.*

Sur un arbre donné, lorsque l'évaluateur démarre sur un nœud donné, il est nécessaire d'avoir la condition de non-circularité de départ afin d'assurer la compatibilité entre la partie de l'évaluateur qui travail sur le sur-arbre et celle qui travaille sur le sous-arbre du nœud de départ.

Dans le cas de la définition FNC, la condition de non-circularité de départ n'est pas nécessaire car l'évaluateur part toujours de la racine qui n'a pas d'attribut hérité.

Dans la condition d'appartenance à la classe FNC, la condition de non-circularité dans le cas d'une grammaire en forme normale peut se réécrire : $R_{FNC}(P, X_0)$ sans circuit.

La différence avec les conditions de clôture classique est que la relation $R_{DNC'}$ n'est pas rajoutée sur les fils de la production de même pour la relation R_{FNC} sur le père. Il n'est pas nécessaire de rajouter la relation $R_{DNC'}$ sur une occurrence de non-terminal fils X_i car le graphe augmenté de $R_{DNC'}(p, X_i)$ contient déjà l'ensemble des dépendances réelles au-dessus de ce fils, dans le contexte de la production p. Si la relation $R_{DNC'}$ est rajoutée sur les fils, on ne profite pas du fait que l'on sait qu'ils ont la production p au-dessus ; des dépendances qui n'existent pas dans le contexte de la production p sont donc rajoutées et elles risquent de créer des circularités. De la même façon il n'est pas nécessaire lors du test FNC de rajouter la relation R_{FNC} du père ; dans ce cas là ce rajout n'est pas gênant car il ne peut pas rajouter de circularité.

L'algorithme DNC′ (cf. tableau 6.1) permet de calculer une famille de relations $\mathcal{R}_{DNC'}(\mathcal{X})$, si la grammaire est DNC′. Avant de lancer cet algorithme il faut vérifier que la grammaire attribuée est FNC et connaître une famille de relations R_{FNC} pour cette grammaire attribuée.

Sur cette classe il est possible de construire un évaluateur de type DNC qui est capable de remonter dans l'arbre.

Cette nouvelle classe est une surclasse de la classe DNC (cf. figure 6.1). Non seulement l'algorithme DNC′ génère moins de dépendances mais le test d'appartenance à la classe est également moins contraignant.

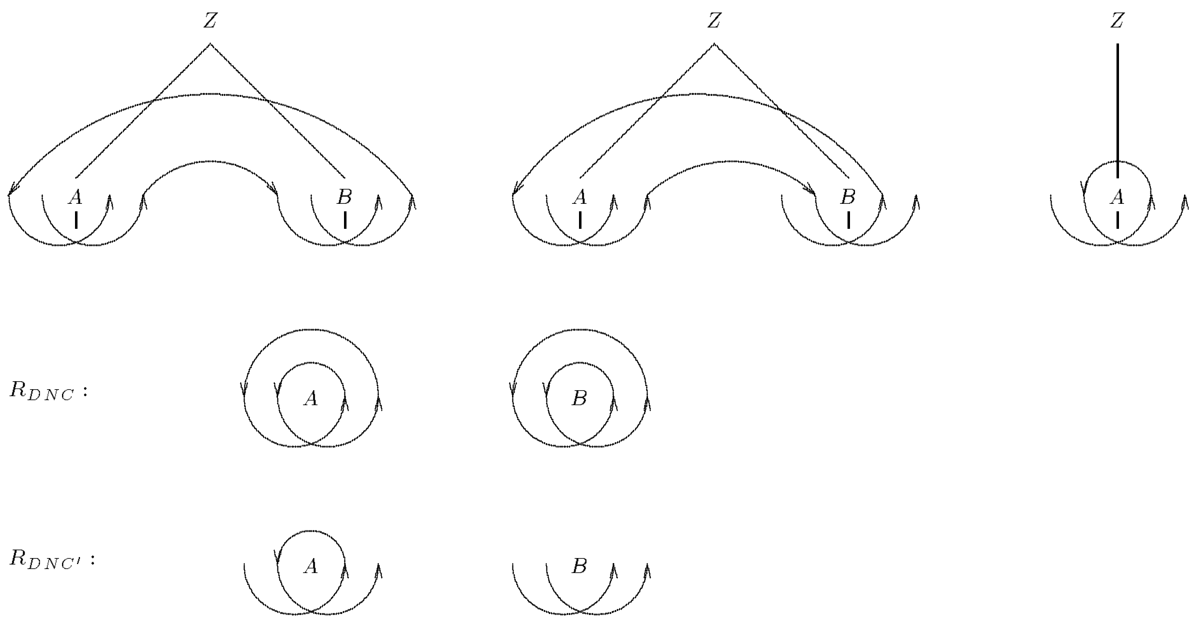


Figure 6.1: Grammaire attribuée DNC′ mais non DNC

6.4 Les classes FNC+ et DNC+

Nous avons vu dans le chapitre précédent que la classe DNC avait été réduite car le fait que l'on connaissait la production au-dessus n'était pas prise en compte. L'idée pour la création de ces nouvelles classes de grammaires attribuées FNC+ et DNC+ est de ne plus associer les relations FNC ou DNC à chaque non-terminal de la grammaire mais à chaque occurrence de non-terminal de chaque production de la grammaire.

/* Calcul des $\mathcal{R}_{DNC'}(\mathcal{X})$ minimaux */

/* On appelle l'algorithme FNC et on génère ainsi la famille de relations d'ordre partiel $\mathcal{R}_{FNC}(\mathcal{X})$ */
Pour chaque $\mathcal{X} \in N$ **faire**
    $\mathcal{R}_{DNC'}(\mathcal{X}) \leftarrow \emptyset$;
Fin pour
Répéter
    `convergence` $\leftarrow$ *vrai* ;
    Pour chaque $\mathcal{X}_0 \in N$ **faire**
        Pour chaque $p : X_0 \to X_1 \cdots X_n$ **faire**
            Pour *pos* **de** 1 **à** n **faire**

$$\mathcal{R}_{DNC'}(p, \mathcal{X}_{pos}) \leftarrow \left[D(p) \cup R_{DNC'}(X_0) \left(\bigcup_{\substack{1 \leq j \leq n \\ j \neq i}} R_{FNC}(X_j) \right) \right]^+ / \mathcal{X}_i \,;$$

                $\mathcal{R} \leftarrow \mathcal{R}_{DNC'}(p, \mathcal{X}_{pos})$;
                Si $\mathcal{R} \not\subseteq \mathcal{R}_{DNC'}(\mathcal{X}_{pos})$ **alors**
                    $\mathcal{R}_{DNC'}(\mathcal{X}_{pos}) \leftarrow \mathcal{R}_{DNC'}(\mathcal{X}_{pos}) \cup \mathcal{R}$;
                    `convergence` $\leftarrow$ *faux* ;
                Fin si
            Fin pour
        Fin pour chaque
    Fin pour chaque
Jusqu'à `convergence`;

/* Test DNC$'$ proprement dit */

/* Appel de test FNC */;
Pour chaque $p : X_0 \to X_1 \cdots X_n$ **faire**
    Pour chaque $i \in \{1, \cdots, n\}$ **faire**
        Si `circulaire`$(R_{DNC'}(p, X_i))$ **alors**
            Stop "La grammaire n'est pas DNC$'$ " ;
        Fin si
    Fin pour chaque;
Fin pour chaque;
Pour chaque $\mathcal{X} \in N$ **faire**
    Si `circulaire`$(\mathcal{R}_{DNC'}(\mathcal{X}) \cup \mathcal{R}_{FNC}(\mathcal{X}))$ **alors**
        Stop "La grammaire n'est pas DNC$'$" ;
    Fin si
Fin pour chaque;
Écrire "La grammaire est DNC$'$ " ;

Tableau 6.1: Algorithme DNC$'$

Chaque relation est donc associée à un contexte particulier. Ce gain d'information permet de raffiner le sélecteur d'arguments et du même coup élargir la classe. En effet, il est ainsi possible de résoudre certaines circularités induites. En raffinant ainsi le sélecteur d'arguments il est également possible de diminuer le nombre de recalculs ou de tests dans le cadre d'un évaluateur FNC ou DNC.

Par contre la taille de la famille de relations est augmentée.

Pour diminuer cette augmentation en taille le raffinement de ce sélecteur d'arguments ne peut être fait que sur une partie des non-terminaux de la grammaire pour résoudre certaines circularités induites.

Il peut être intéressant de pousser l'association de la relation à un contexte plus important, c'est-à-dire à un morceau d'arbre. Nous pouvons ainsi imaginer résoudre de nouvelles circularités.

D'un point de vue théorique cela ne change rien. Pour cela il suffit de considérer un morceau d'arbre comme une nouvelle production dont les règles sémantiques sont construites à partir des règles sémantiques associées aux différentes instances productions constituant le morceau d'arbre.

Nous allons donc présenter un algorithme où une relation est attachée à chaque occurrence de non-terminal de chaque production. Pour chaque production $p : X_0 \to X_1 \cdots X_n$, une relation $R_{FNC+}(\mathcal{X}_0, 0, p) \subseteq H(X_0) \times S(X_0)$ et une famille de relations $(R_{DNC+}(\mathcal{X}_i, i, p))_{1 \leq i \leq n}$ chacune incluse dans $S(X_i) \times H(X_i)$ sont construites. Nous présentons l'algorithme qui calcule la famille des R_{FNC+} minimale (cf tableau 6.2) et celui qui calcule la famille des R_{DNC+} minimale (cf tableau 6.3).

Pour alléger les algorithmes nous avons utilisé la notation :
$(\mathcal{X}, i, p)$ pour parler du non-terminal $\mathcal{X}$ qui apparaît en position i dans la production p.

Simplification et optimisation des algorithmes Plutôt que d'ajouter les relations $R_{FNC+}(\mathcal{X}_i, 0, p_i)$ sur tous les fils de la production, il peut être intéressant de ne rajouter que la relation $R_{FNC}(\mathcal{X}_i)$ sur certains fils. Il n'est alors pas nécessaire de parcourir toutes les combinaisons de couples occurrence de non-terminal, production.

En raffinant un peu cette idée, on peut associer une relation à une occurrence de non-terminal et à une partie des productions dont il est le père. Le nombre de combinaison de couples est ainsi réduit.

Il paraît intéressant également de regarder des propriétés de couverture [19] entre les relations pour éviter de faire certains tests.

Il est aussi possible d'appliquer ces simplifications ou optimisations au test DNC+.

L'évaluateur de type FNC+ (resp. DNC+) est comparable à l'évaluateur de type FNC (resp. DNC), sauf que connaissant la production courante, il tient compte du sélecteur d'arguments associé à cette production.

Conclusion

Les classes de grammaires attribuées ainsi obtenues sont plus larges que les classes FNC et DNC. Il est très simple de généraliser les résultats des classes FNC et DNC à ces

/* Calcul des $R_{FNC+}(\mathcal{X}, 0, p)$ minimaux */

Pour chaque $(\mathcal{X}, 0, p)$ **faire**
  $R_{FNC+}(\mathcal{X}, 0, p) \leftarrow \emptyset$;
Fin pour chaque
Répéter
  `convergence` $\leftarrow$ *vrai* ;
  Pour chaque $(\mathcal{X}, 0, p)$ **faire** /* avec $p : X_0 \rightarrow X_1 \cdots X_n$ */
    Pour chaque $((\mathcal{X}_1, 0, p_1), \cdots, (\mathcal{X}_n, 0, p_n))$ **faire**tt /* Pour toutes les combinaisons */
      $R \leftarrow \left[D(p) \cup \left(\bigcup_{1 \leq j \leq n} R_{FNC+}(\mathcal{X}_j, 0, p_j) \right) \right]^+ / X_0$;
      Si $R \not\subseteq R_{FNC+}(\mathcal{X}, 0, p)$ **alors**
        $R_{FNC+}(\mathcal{X}, 0, p) \leftarrow R_{FNC+}(\mathcal{X}, 0, p) \cup R$;
        `convergence` $\leftarrow$ *faux* ;
      Fin si
    Fin pour chaque
  Fin pour chaque
Jusqu'à `convergence`;

/* Test FNC+ proprement dit */

Pour chaque $p : X_0 \rightarrow X_1 \cdots X_n$ **faire**
  Pour chaque $((\mathcal{X}_1, 0, p_1), \cdots, (\mathcal{X}_n, 0, p_n))$ **faire**
    Si `circulaire` $\left(\left[D(p) \cup \left(\bigcup_{1 \leq j \leq n} R_{FNC+}(\mathcal{X}_j, 0, p_j) \right) \right]^+ / X_0 \right)$ **alors**
      Stop "La grammaire n'est pas FNC+" ;
    Fin si
  Fin pour chaque
Fin pour chaque
Écrire "La grammaire est FNC+" ;

Tableau 6.2: Algorithme FNC+

/* Calcul des $R_{DNC+}(\mathcal{X}, 0, p)$ minimaux */

On suppose la famille R_{FNC+} construite.

Pour chaque $(\mathcal{X}, i, p)$ **avec** $i \neq 0$ **faire**
    $R_{DNC+}(\mathcal{X}, i, p) \leftarrow \emptyset$;
Fin pour chaque
Répéter
    `convergence` $\leftarrow$ *vrai* ;
    Pour chaque $(\mathcal{X}, i, p)$ **avec** $i \neq 0$ **faire**
        Pour chaque $((\mathcal{X}_0, j, p_0), \cdots, (\mathcal{X}_k, 0, p_k), \cdots)$ **avec** $1 \leq k \leq n, k \neq i$ et $j \neq 0$ **faire**

$$R \leftarrow \left[D(p) \cup R_{DNC+}(\mathcal{X}_0, j, p_0) \cup \left(\bigcup_{\substack{1 \leq k \leq n \\ k \neq i}} R_{FNC+}(\mathcal{X}_k, 0, p_k) \right) \right]^+ / X_i \,;$$

            Si $R \not\subseteq R_{DNC+}(\mathcal{X}, i, p)$ **alors**
                $R_{DNC+}(\mathcal{X}, i, p) \leftarrow R_{DNC+}(\mathcal{X}, i, p) \cup R$;
                `convergence` $\leftarrow$ *faux* ;
            Fin si
        Fin pour chaque
    Fin pour chaque
Jusqu'à `convergence`;

/* Test DNC+ proprement dit */

Pour chaque $(\mathcal{X}, i, p)$ **avec** $i \neq 0$ **faire**
    Pour chaque $((\mathcal{X}_0, j, p_0), \cdots, (\mathcal{X}_k, 0, p_k), \cdots)$ **avec** $1 \leq k \leq n, k \neq i$ et $j \neq 0$ **faire**

$$R \leftarrow \left[D(p) \cup R_{DNC+}(\mathcal{X}_0, j, p_0) \cup \left(\bigcup_{\substack{1 \leq k \leq n \\ k \neq i}} R_{FNC+}(\mathcal{X}_k, 0, p_k) \right) \right]^+ / X_i \,;$$

        Si `circulaire`(R) **alors**
            Stop "La grammaire n'est pas DNC+" ;
        Fin si
    Fin pour chaque
Fin pour chaque
Pour chaque $\mathcal{X} \in N$ **faire**
    Pour chaque $((\mathcal{X}, 0, p), (\mathcal{X}, i, p')), i \neq 0$ **faire**
        Si `circulaire`$(R_{FNC+}(\mathcal{X}, 0, p) \cup R_{DNC+}(\mathcal{X}, i, p'))$ **alors**
            Stop "La grammaire n'est pas DNC+" ;
        Fin si
    Fin pour chaque
Fin pour chaque
Écrire "La grammaire est DNC+" ;

Tableau 6.3: Algorithme DNC+

nouvelles classes. Si les relations sont associées à des morceaux d'arbres et en supposant que ces morceaux d'arbres sont tous les arbres possibles, on revient au test NC. Une analogie est possible entre ces nouvelles classes et la famille de classes LR(k) pour les grammaires, en effet on va chercher un peu plus d'information dans l'arbre source pour résoudre un problème.

Nous pouvons espérer trouver, à partir d'un test de circularité FNC, une heuristique qui essaye d'élargir un peu la classe uniquement sur les non-terminaux ou productions où les circularités apparaissent afin de les éliminer. Je n'ai pas eu le temps de me pencher sur ce genre de recherche.

Chapitre 7

Réutilisabilité

7.1 La généricité dans les grammaires attribuée

Un gène est une grammaire attribuée qui spécifie un algorithme qui a des chances d'être réutilisé ; cet algorithme correspond à une résolution possible d'un problème précis. Cette résolution doit être la meilleure aussi bien au niveau de la généralité qu'au niveau de l'efficacité. Pour rendre ce gène le plus général possible, la syntaxe d'entrée du gène doit être réduite aux entités syntaxiques qui jouent un rôle dans la résolution du problème. Les attributs et les règles sémantiques définissant ceux-ci doivent être réduits au minimum.

Pour instancier la sémantique d'un gène sur une grammaire réelle, il faut qu'à chaque arbre de la grammaire réelle soit associé un arbre pour la syntaxe de base du gène. Cette association s'effectue grâce à une grammaire attribuée fonctionnelle, appelée grammaire attribuée de couplage, qui peut être totalement construite par l'utilisateur comme dans l'approche de Farrow [5] ou d'une façon semi-automatique comme dans notre approche [41].

Il est alors possible d'appliquer la méta-composition pour plonger les règles sémantiques du gène sur la grammaire réelle. Cette approche permet ensuite de modifier ou de compléter la spécification obtenue. Si on est sûr de la validité du résultat obtenu ou que l'on veut avoir la compilation séparée des évaluateurs, il est également possible d'appliquer la composition des évaluateurs.

Nous allons présenter ici une approche très simple de la construction de la grammaire attribuée de couplage qui fait passer d'un arbre d'une grammaire réelle vers un arbre de la grammaire de base d'un gène.

Cette construction est très simple car la grammaire attribuée ainsi construite est purement synthétisée, sans If_Then_Else et les différents algorithmes ne recherchent pas l'optimalité.

Nous avons choisi de construire une grammaire attribuée fonctionnelle purement synthétisée car d'après [39] nous savons que la composition d'une grammaire attribuée purement synthétisée avec une grammaire attribuée FNC est FNC.

On peut trouver une présentation un peu différente de cette construction dans [9].

Construction automatique de la grammaire attribuée de couplage

Notre approche consiste à laisser à l'utilisateur la tâche de spécifier une association entre les terminaux et les non-terminaux des deux grammaires. Il doit trouver quels sont les non-terminaux et les terminaux qui ont un rôle dans l'algorithme décrit par le gène et ensuite associer ceux-ci avec l'entité correspondante dans la grammaire du gène. Nous pensons que cette association est suffisante pour pouvoir construire la grammaire attribuée de couplage.

Pour plus de facilité nous parlons de grammaire d'entrée pour la grammaire réelle et de grammaire de sortie pour la grammaire du gène.

Pour décrire l'association l'utilisateur doit construire une application de correspondance. Cette application doit suivre certaine règles.

Premièrement, chaque non-terminal (resp. terminal) de la grammaire de sortie doit être associé avec au moins un non-terminal (resp. terminal) de la grammaire d'entrée. Deuxièmement, à chaque non-terminal (resp. terminal) de la grammaire d'entrée est associé au plus un non-terminal (resp. terminal) de la grammaire de sortie. Finalement nous imposons que les racines des deux grammaires soient associées ensembles.

Définition 37 (Application de Correspondance) *Une* application de correspondance Cor *entre la grammaire d'entrée* $G_i = (N_i, T_i, Z_i, P_i)$ *et la grammaire de sortie* $G_o = (N_o, T_o, Z_o, P_o)$ *se définit grâce à deux applications* Cor_T *et* Cor_N

$$Cor_N : N_o \to \bar{\mathcal{P}}(N_i),^1$$
$$Cor_T : T_o \to \bar{\mathcal{P}}(T_i)$$

avec

$$Cor : N_o \cup T_o \to \bar{\mathcal{P}}(N_i) \cup \bar{\mathcal{P}}(T_i)$$
$$Cor(x) = \begin{cases} Cor_N(x) & \text{si } x \in N_o \\ Cor_T(x) & \text{si } x \in T_o \end{cases}$$

telle que

- $\forall x, y \in (N_o \cup T_o), x \neq y \Rightarrow Cor(x) \cap Cor(y) = \emptyset$ *;*
- $Cor(\mathcal{Z}_o) = \{Z_i\}$.

Nous présenterons un peu plus loin comment relaxer certaines de ces contraintes.

Grâce à l'application Cor et à la relation classique de "dérivation" $\Longrightarrow$ sur G_i, il est possible de définir différents ensembles de terminaux ou non-terminaux.

Définition 38 (Terminaux, non-terminaux clefs, neutres et morts) *Soient* G_i, G_o *et* Cor *définis comme ci-dessus, nous avons :*

[1] $\bar{\mathcal{P}}(N)$ représente l'ensemble des sous-ensembles *non vides* de N.

- N_k=$\{\mathcal{X} \in N_i \mid \exists \mathcal{Y} \in N_o, \mathcal{X} \in Cor(\mathcal{Y})\}$ *est l'ensemble des* non-terminaux clefs *;*
- T_k=$\{\mathcal{X} \in T_i \mid \exists \mathcal{Y} \in T_o, \mathcal{X} \in Cor(\mathcal{Y})\}$ *est l'ensemble des* terminaux clefs *;*
- N_n=$\{\mathcal{X} \in N_i - N_k \mid \exists \mathcal{Y} \in N_k, \mathcal{X} \stackrel{*}{\Longrightarrow} \mathcal{Y}\}$ *est l'ensemble des* non-terminaux neutres *;*
- $N_d = N_i - (N_k \cup N_n)$ *est l'ensemble des* non-terminaux morts *;*
- $T_d = T_i - T_k$ *est l'ensemble des* terminaux morts*.*

Il n'y a pas de terminal neutre.

Réduction en arbre clos

A partir de l'association entre les symboles des grammaires définie par Cor, nous désirons définir une association entre les arbres. L'idée est que seul les symboles clefs sont importants, c'est pourquoi nous décomposons un arbre d'entrée en *(sous-)arbres clos*, i.e. sous-arbres délimités par des symboles clefs.

Pour construire l'ensemble des arbres clos, nous transformons la grammaire d'entrée. La première étape consiste à oublier les symboles morts (ceux de T_d et N_d), qui ne jouent aucun rôle dans le couplage. Ils sont éliminés ainsi que les productions où ils apparaissent en partie gauche.

Pour plus de simplicité, plus loin, nous appelons toujours la grammaire résultante G_i, avec $T_i = T_k$ et $N_i = N_k \cup N_n$.

Définition 39 (Arbre clos) *Soient G_i, G_o et Cor définis comme précédemment, un arbre clos est un arbre de G_i tel que :*

- *la racine a son type dans N_k ;*
- *les feuilles ont leur type dans $N_k \cup T_k$;*
- *les nœud internes ont leur type dans N_n.*

Supposons que chaque arbre clos est associé à un sous-arbre de sortie correspondant de la grammaire de sortie ; par "correspondant" nous entendons qu'il a la même frontière, suivant le mapping défini par Cor.

Ensuite, ces sous-arbres de sortie sont réunis conformément à la façon dont les arbres clos étaient réunis dans l'arbre d'entrée ; cela donne l'arbre de sortie correspondant.

Pour être certain que ce processus définit une fonction, et pour être sûr que nous pouvons construire cette fonction de façon statique, nous imposons que le nombre ainsi que la taille des arbres clos de G_i soit fini.

La deuxième transformation appliquée à G_i consiste à éliminer les non-terminaux neutres récursifs, pour être sûr de n'avoir que des arbres clos de taille finie. Plus précisément, nous recherchons les non-terminaux neutres qui dérivent dans eux-mêmes sans passer par un non-terminal clef.

C'est-à-dire, si $\mathcal{X} \in N_k$, $\mathcal{Y} \in N_n$ et P_i contient les productions $\mathcal{X} \to \mathcal{Y}$, $\mathcal{Y} \to \mathcal{Y}\,\mathcal{Y}$ et $\mathcal{Y} \to \mathcal{X}$. Alors le nombre, tout comme la taille, des arbres clos qu'il est possible de construire à partir de $\mathcal{X}$ et de $\mathcal{Y}$ avec des $\mathcal{X}$ comme feuilles est non borné. Il est donc nécessaire de rajouter une condition afin de ne jamais rencontrer ce type de situation. Nous donnons maintenant l'algorithme qui vérifie cette condition et qui transforme G_i pour que la construction de l'ensemble des arbres clos soit plus simple.

A cet effet, nous définissons une nouvelle relation de dérivation, notée $\underset{Cor}{\Longrightarrow}$. Celle-ci est une restriction de $\Longrightarrow$, telle que $\mathcal{X} \underset{Cor}{\Longrightarrow} \mathcal{Y}$ si et seulement si $\mathcal{X} \Longrightarrow \mathcal{Y}$ et $\mathcal{X} \notin N_k$.

L'algorithme d'élimination des récursions est le suivant :

Algorithme d'élimination des récursions

1. En utilisant la définition de la relation de dérivation, nous générons l'ensemble $N_r = \{\mathcal{X} \in N_n \mid \mathcal{X} \overset{*}{\underset{Cor}{\Longrightarrow}} \mathcal{X}\}$.

2. Sur cet ensemble N_r, nous construisons les classes d'équivalence pour la relation Γ définie par :
$$\mathcal{X}\ \Gamma\ \mathcal{Y} \Longleftrightarrow (\mathcal{X} \overset{*}{\underset{Cor}{\Longrightarrow}} \mathcal{Y} \text{ et } \mathcal{Y} \overset{*}{\underset{Cor}{\Longrightarrow}} \mathcal{X})$$
Nous renommons chaque non-terminal de N_r par le nom de sa classe d'équivalence.[2].

3. Dans cette nouvelle grammaire, les non-terminaux neutres dérivent dans eux-mêmes uniquement par des productions de la forme $\mathcal{X} \to \mathcal{X}$ avec $\mathcal{X} \in N_r$, sinon nous considérons que le couplage n'est correct.

4. Ensuite, nous éliminons les production de ce type de G_i.

Dans cette nouvelle grammaire G_i, il n'y a plus de récursion sur les non-terminaux neutres. Nous pouvons alors commencer la construction des arbres clos. Pour chaque $\mathcal{X} \in N_k$, nous voulons générer tous les arbres clos possible de G_i avec pour racine $\mathcal{X}$. Pour ce faire, nous utilisons un algorithme de point fixe appelé *algorithme de clôture d'arbre*, qui manipule des arbres (incomplets) de G_i. Sur chaque arbre de cet ensemble nous appliquons une transformation qui génère un nouvel ensemble, ceci jusqu'à ce que l'ensemble final ne contienne plus que des arbres clos. La transformation est composée de pas élémentaires, notés $t \xrightarrow{u,p} t'$, qui, étant donné un arbre t, une feuille u de t et une production p de G_i telle que $label(u) = LHS(p)$[3] $\in N_n$, remplace u par l'arbre $p(u_1, \ldots, u_n)$. L'ensemble de départ est constitué des productions dont la partie gauche fait partie N_k. L'algorithme est présenté dans la figure 7.1. Comme tous les non-terminaux neutres récursifs ont été éliminés, nous sommes sûrs que l'algorithme termine.

[2] Ce renommage est aussi appliqué sur P_i.

[3] Left Hand Side de p.

$R' \leftarrow \{t \mid \exists p \in P_i, LHS(p) \in N_k, t = p(u_1, \ldots, u_n)\}$;
Répeter
  $R \leftarrow R'$
  $R' \leftarrow \emptyset$;
  Pour chaque arbre t de R
    Si t est un arbre clos **alors**
      $R' \leftarrow \{t\} \cup R'$
    Sinon
      Pour chaque feuille u de t avec $label(u) = \mathcal{X} \in N_n$
        Pour chaque production p avec $\mathcal{X} = LHS(p)$
          $t \xrightarrow{u,p} t'$;
          $R' \leftarrow \{t'\} \cup R'$;
Jusqu'à $R = R'$.

Figure 7.1: Algorithme de clôture d'arbre

Instanciation

Maintenant, notre but est de déterminer, pour chaque arbre clos, s'il existe un sous-arbre de la grammaire de sortie qui lui correspond. Nous avons choisi une approche très simple, dans laquelle les sous-arbres de sortie ont une hauteur un, c'est-à-dire que ce sont des instances de productions. Nous étendons plus loin cette construction à des arbres plus complexes.

A chaque arbre clos t, nous associons sa *production virtuelle d'entrée* p ; elle a pour forme $X_0 \rightarrow Y_1 \ldots Y_n$, où $\mathcal{X}_0$ est le label de la racine de t et les $\mathcal{Y}_i$ $(1 \leq i \leq n)$ sont de gauche à droite les labels des feuilles de t $(\mathcal{Y}_i \in (N_k \cup T_k))$.

A cette production virtuelle d'entrée nous associons une *production virtuelle de sortie*, qui est son image par Cor, c'est-à-dire que chaque symbole $\mathcal{X} \in N_k \cup T_k$ de la production virtuelle d'entrée est remplacé par $\mathcal{Y}$ avec $X \in Cor(\mathcal{Y})$ pour donner la production virtuelle de sortie.

Dans cette approche simple, nous testons si la production virtuelle de sortie est effectivement une production de G_o. Si ce n'est pas le cas, nous ne savons pas quel arbre de sortie associer à l'arbre clos et nous considérons que le couplage Cor n'est pas correct.

Cette approche est évidemment très restrictive, mais nous pensons quand même pouvoir traiter beaucoup de cas pratiques.

Le résultat à cette étape de la construction est un ensemble de paires (p_o, t) où p_o est dans P_o et t est un arbre clos de G_i. Nous pouvons dire maintenant que la correspondance entre les deux grammaire est correcte. Cet ensemble de paires définit une fonction des arbres de G_i vers (un sous-ensemble) des arbres de G_o et il est facile de voir comment l'implanter de façon abstraite :

1. découper l'arbre d'entrée en arbres clos au niveau des symboles clef ;

2. pour chaque arbre clos, le remplacer par la production de la grammaire de sortie qui

lui correspond. Ce remplacement s'effectue à l'enplacement où se trouvait l'arbre clos dans l'arbre d'entrée et en ajoutant un lien de chaque nœud de cette production avec les symboles clefs correspondants dans l'arbre d'entrée.

Tous les calculs d'un arbre de sortie peuvent maintenant être transposés sur l'arbre correspondant d'entrée.

7.1.1 Génération Automatique de la Grammaire Attribuée de Couplage

Dans cette section nous voulons un couplage correct sous la forme d'une grammaire attribuée fonctionnelle qui effectuera la traduction des arbres de G_i vers les arbres de G_o conformément au couplage.

Génération de la Grammaire Attribuée de Couplage

La construction de la grammaire attribuée de couplage commence quand la validation du couplage est terminée. Nous voulons construire une grammaire attribuée fonctionnelle purement synthétisée $\alpha = (G_i, A_\alpha, F_\alpha)$ qui pour chaque paire (p_o, t) effectue la transformation de t vers p_o.

Pour rendre la construction de α plus claire, nous décomposons notre algorithme en deux étapes. Premièrement nous construisons des pseudo grammaires attribuées pour chaque paire (p_o, t). Ensuite nous vérifions que ces différentes pseudo grammaires attribuées sont compatibles les unes avec les autres afin de construire correctement α.

Chacune de ces étapes peut échouer, même si le couplage est correct.

Contrainte initiale

Il existe tout d'abord une contrainte importante sur les non-terminaux clefs $\mathcal{X}$, induite par l'application Cor :

$\mathcal{X}$ doit avoir exactement un attribut de type $Cor^{-1}(\mathcal{X})$.

Pour plus de facilité nous nommerons l'attribut par son type.

Construction d'une pseudo grammaire attribuée pour une paire (p_o, t)

Nous désirons obtenir une grammaire attribuée de couplage qui effectue sur un nœud u de l'arbre t de chaque paire (p_o, t), la construction de p_o. Au-dessus de u, il n'y a que des règles de copie vers la racine ; au-dessous de u les arguments de la construction sont amenés depuis les feuilles de t par des règles de copie. En général il y a plusieurs choix possibles pour u.

Afin de les déterminer nous appliquons un algorithme bottom-up sur t. Le pas élémentaire sur chaque nœud u consiste à décider si sur ce nœud il est possible d'effectuer la construction. Si ce n'est pas possible toutes les occurrences d'attributs qui se trouvent sur la partie droite de la production sont passées à la partie gauche.

Dans certains cas, le passage des attributs vers la partie gauche induit sur la partie gauche la déclaration de deux attributs de même nom (de même type). C'est pourquoi nous avons introduit la notion d'attribut indexé, noté (a, j), qui remplace les attributs classiques. L'index j permet de distinguer deux attributs de même nom sans perdre leur ordre relatif.

Pour chaque t et chaque $\mathcal{X} \in N_i$ nous avons un ensemble $A_t(\mathcal{X})$ d'attributs indexés (a, j), leur type est $a \in (N_o \cup T_o)$. Pour chaque t et chaque $p \in P_i$ nous avons un ensemble $F_t(p)$ de règles sémantiques entre attributs indexés, noté $r_{p,(a,j)}$.[4]

Dans l'arbre clos t nous ne faisons pas la différence entre les non-terminaux clefs et les terminaux clefs. La contrainte initiale devient alors :

$$\forall \mathcal{X} \in N_k \cup T_k, A_t(\mathcal{X}) = \{(a, 1)\} \text{ avec } \mathcal{X} \in Cor(a).$$

Afin de simplifier l'algorithme, nous ne testons pas l'unicité des éléments de $F_t(p)$; il est clair que la non-unicité implique l'impossibilité de construire α.

Maintenant nous présentons le pas élémentaire sur le nœud u avec $p = prod(u)$. Dans cette version de l'algorithme nous faisons toujours la construction sur la production racine. Le pas élémentaire utilise deux fonctions, $AP(p)$ qui construit un ensemble d'attributs indexés AP pour $LHS(p)$ et $FP(p)$ qui construit un ensemble de règles sémantiques FP sur p et un prédicat *Construction(p)* qui teste si la construction de p_o est possible sur p. AP, FP et *Construction* sont décrits dans la figure 7.2.

D'après la définition de (p_o, t) et la contrainte initiale, le nombre total d'attributs indexés sur les feuilles de t est égal à la longueur de la partie droite de p_o. Dès que nous avons accès à ce nombre d'occurrences d'attributs sur une production, nous savons qu'il est possible d'effectuer la construction.

Pas élémentaire sur un nœud u de t

Nous notons r la racine de t et $\{(a, 1)\}$ le singleton associé $A_t(label(r))$. Alors,

1. Si u est la racine de t :

   (a) Si *Construction(prod(u))* est vrai :

   $$F_t \leftarrow F_t \cup \{r_{prod(u),(a,1)} : X_0.(a, 1) = p_o(\dots)\}$$

   (b) Sinon la construction de α échoue.

2. Si $A_t(label(u)) = \emptyset$:

   $$A_t(label(u)) \leftarrow AP(prod(u)) ; F_t \leftarrow F_t \cup FP(prod(u))$$

[4] Nous oublions la position du non-terminal car celle-ci est toujours égale à zéro comme α purement synthétisée.

Fonction $AP(p)$
$p : X_0 \rightarrow X_1 \dots X_n$
$AP \leftarrow \emptyset$
Pour chaque $\mathcal{X}_i \in RHS(p)$
Pour chaque $(x, k) \in A_t(\mathcal{X}_i)$
$AP \leftarrow AP \cup \{(x, Card(x, AP) + 1)\}$
où $Card(x, AP)$ renvoie le nombre d'attributs indexés (x, j) dans AP
qui ont un même type x

Fonction $FP(p)$
$p : X_0 \rightarrow X_1 \dots X_n$
$FP \leftarrow \emptyset$
Pour chaque x, tel que $\exists(x, k) \in A_t(\mathcal{X}_0)$
Pour chaque k pris dans l'ordre croissant
$FP \leftarrow FP \cup \{r_{p,(x,k)} : X_0.(x, k) = X_i.(x, j)\}$
où l'occurrence d'attribut $X_i.(x, j)$ est "la plus à gauche" [5]
qui ne soit pas utilisée en partie gauche d'une règle de copy qui est déjà dans $FP(p)$.

Prédicat *Construction(p)*
Le cardinal de *AP(p)* est égal au nombre de symboles en partie droite de p_o.

(i,j) est le plus petit couple dans l'ordre lexicographique.

Figure 7.2: Les fonctions AP, FP et le prédicat *Construction*.

3. Si $A_t(label(u)) \neq \emptyset$:

   (a) Si $AP(prod(u)) = A_t(label(u))$:

   $$F_t \leftarrow F_t \cup \{FP(prod(u)\}$$

   (b) Sinon la construction de α échoue.

Nous rappelons que la construction de α échoue également si plusieurs règles sémantiques définissant une même occurrence d'attribut d'une même production sont ajoutées à F_t.

Notre algorithme donne l'unique solution pour laquelle la construction de p_o se fait sur la production racine. En fait, il est facile de modifier ce pas élémentaire afin de produire un ensemble de solutions. Chaque solution correspond à une position dans l'arbre où la construction de p_o est possible. Nous discutions de cette possibilité dans la partie 7.1.2.

Instanciation finale

L'étape finale consiste à regrouper tous les A_t, F_t de façon correcte, afin d'obtenir les ensembles A_α et F_α qui définissent une grammaire attribuée fonctionnelle correcte.

Pour chaque non-terminal, nous vérifions qu'il porte les mêmes attributs indexés dans chaque A_t. Pour chaque production, nous vérifions qu'elle porte les mêmes règles sémantiques dans chaque F_t.[6] Si tel est le cas, la construction de la grammaire attribuée de couplage est possible et consiste ni plus ni moins à faire l'union des différents A_t et F_t. La seule modification à apporter concerne les non-terminaux neutres récursifs qui ont été éliminés dans l'algorithme de validation. Ces non-terminaux ont été renommés par leur classe d'équivalence. Pour chacune de ces classes il existe un unique ensemble d'attributs déclarés. Sur chaque non-terminal d'une même classe nous déclarons l'ensemble des attributs indexés de la classe. Sur chaque production éliminée à l'étape 4 de l'algorithme d'élimination des récursions, nous générons des règles de copie entre les attributs en partie gauche et ceux en partie droite. C'est la seule sémantique acceptable pour de telles productions.

Si cette vérification réussit, nous avons une déclaration consistante sur chaque élément de $N_i \cup T_i$ et un ensemble de règles sémantiques sur chaque production de G_i. Maintenant nous devons transformer ces ensembles pour obtenir une grammaire attribuée classique avec des vrais attributs. De plus, la grammaire de base doit être G_i avec ses non-terminaux et terminaux morts. Pour les non-terminaux nous renommons l'attribut indexé (a, i) en $a.i$ dans les déclarations et les règles sémantiques. Pour les terminaux T_i, nous remplaçons $(a, 1)$ dans les règles sémantiques par ϵ.

A la fin de cette construction, la grammaire attribuée de couplage à les propriétés suivantes :

1. elle génère un arbre de G_o et pas une forêt ;

[6]Nous ne tenons pas compte des ensembles vides.

2. chaque occurrence d'attribut est utilisée une fois et une seule sur une production, l'arbre construit ne contient pas de dag ;

3. elle est purement synthétisée.

7.1.2 Extensions de l'algorithme de base

Solutions multiples

Dans l'algorithme de base nous présentons une unique solution pour la position de construction de p_o dans l'arbre clos ; celle-ci est faite à la racine de l'arbre clos. Il est en fait possible d'effectuer la construction plus bas dans l'arbre clos. La condition pour pouvoir effectuer la construction sur un nœud est donnée par le prédicat *Construction*.

Pour un arbre clos donné t nous avons maintenant un ensemble $S(t)$ de solutions acceptables. Nous essayons alors de trouver une solution dans chaque $S(t)$ qui soit compatible avec les autres $S(t')$. En fait, au pire, nous essayons toutes les combinaisons possibles.

Permutation

Dans l'algorithme de base nous imposons que chaque production virtuelle de sortie soit exactement une production de P_o ; ce qui signifie en particulier que l'ordre des symboles en partie droite n'est pas modifié.

Nous aurions pu choisir une stratégie moins contraignante, par exemple que pour chaque production virtuelle de sortie p il existe dans P_o une production p' dont la partie droite est une permutation de la partie droite de p. Ceci peut, le cas échéant, introduire des ambiguïtés dans le choix de p'. Il est facile de les résoudre en imposant un ordre sur les permutations et en choisissant la plus petite qui soit valide.

Contrainte sur la racine

Nous cherchons ici à relaxer la contrainte $Cor(\mathcal{Z}_o) = \{\mathcal{Z}_i\}$. En fait pour un arbre donné t de G_i, nous voudrions pouvoir instancier un même gène sur différents sous-arbres de t.

Nous acceptons donc de générer une grammaire attribuée de couplage α qui construise une forêt telle que le type de chaque arbre construit est $\mathcal{Z}_o$. Pour cela nous devons vérifier la condition suivante :

$$\forall \mathcal{X} \in N_k \ tel \ que \ \mathcal{Z}_i \underset{Cor}{\overset{*}{\Longrightarrow}} \mathcal{X}, \mathcal{X} \in Cor(\mathcal{Z}_o)$$

Cette nouvelle approche implique une nouvelle définition pour les non-terminaux neutres :

$$N_n = \{\mathcal{X} \in N_i - N_k \mid \exists \mathcal{Y} \in N_k - Cor(\mathcal{Z}_o), \mathcal{X} \overset{*}{\Longrightarrow} \mathcal{Y}\}$$

De plus, nous ajoutons à l'étape d'élimination des symboles morts, l'élimination des non-terminaux inclus dans $Cor(Z_o)$ qui apparaissent en partie droite d'une production. Ceci assure que les arbres de sortie ne sont pas reliés les uns aux autres. Sinon l'algorithme de construction de la grammaire de couplage reste inchangé.

Extension de instanciation

Nous avons décrit dans l'instanciation l'association d'un arbre clos avec une unique production, nous étendons cette association en permettant l'association d'un arbre clos avec un sous-arbre de hauteur supérieure à un.

Nous construisons comme avant *la production virtuelle de sortie* associée à l'arbre clos. Ensuite, sa partie droite est vue comme une phrase de G_o dont l'arbre de dérivation sera associé à l'arbre clos. La construction de l'arbre de dérivation est simple grâce à un parser légèrement transformé.

La deuxième étape consiste à construire la grammaire attribuée de couplage. Il facile de modifier l'algorithme de la section 7.1.1 si celui-ci construit tout l'arbre sur une seule production. Si la construction est répartie dans l'arbre, cette transformation semble plus complexe.

La troisième étape qui consiste à regrouper les pseudo grammaires attribuées n'est pas modifiée.

7.1.3 Remarques

Notre but était de libérer l'utilisateur des détails de la tache de spécification de la grammaire attribuée couplage. Nous croyons que les algorithmes présentés ici remplissent bien ce but, par contre il nous reste à les implanter et à vérifier que le pouvoir d'expression est suffisant.

Ces algorithmes constituent bien sûr une première approche. Celle-ci a été en partie complétée dans [9].

7.2 L'extraction automatique de gènes

7.2.1 Introduction

Soit une grammaire attribuée, supposons que dans celle-ci nous traitons un problème particulier, par exemple la création d'une table des symboles. Ce traitement nous intéresse pour une autre application, mais dans cette première grammaire attribuée, celui-ci est mélangé avec le traitement d'autres problèmes. Il paraît intéressant de pouvoir extraire ce problème précis de la grammaire attribuée de départ, soit en gardant le même support syntaxique et en générant de nouvelles règles sémantiques afin d'obtenir une nouvelle grammaire attribuée, soit en essayant d'avoir une syntaxe minimale, sans perdre en généralité, pour pouvoir utiliser cette grammaire comme un gène. Nous proposons ici une série d'algorithmes qui permettent de faire ce travail de façon automatique.

7.2.2 Simplification de la sémantique

Pour simplifier les algorithmes nous supposerons que la solution de notre problème est contenue dans un seul attribut donné $b \in \mathcal{A}(\mathcal{B})$. En prenant ce choix, les grammaires attribuées sont vues comme des fonctions, mais il facile de généraliser les algorithmes à plusieurs attributs de sortie. Il est également possible de vouloir spécifier des occurrences d'attributs d'entrée en plus de l'attribut de sortie pour obtenir une GAI.

Nous allons commencer par repérer toutes les occurrences d'attributs nécessaires au calcul de l'attribut résultat. A chaque attribut $a \in \mathcal{A}(\mathcal{X})$ un booléen noté, $\mathcal{X}.a.B$ est associé.

$\forall \mathcal{X} \in N, \forall a \in \mathcal{A}(\mathcal{X}), \mathcal{X}.a.B \leftarrow faux$;
$\mathcal{B}.b.B \leftarrow vrai$;
Répéter
  convergence $\leftarrow$ *vrai* ;
  Pour chaque $p \in P, p : X_0 \to X_1 \ldots X_n$ **faire**
    Pour chaque $(X_i.a_1, X_j.a_2) \in D(p)$ **faire**
      Si $X_i.a_1$ *n'est pas une occurrence d'attribut d'entrée* **alors**
        Si $(\mathcal{X}_i.a_1.B = vrai$ **et** $\mathcal{X}_j.a_2.B = faux)$ **alors**
          $\mathcal{X}_j.a_2.B \leftarrow vrai$;
          convergence $\leftarrow$ *faux* ;
        Fin si
      Fin si
    Fin pour
  Fin pour
Jusqu'à *convergence.*

Après passage de l'algorithme, tous les attributs dont le booléen associé est resté à faux ne servent jamais au calcul de l'attribut de sortie. Toutes les déclarations de ces attributs et toutes les règles sémantiques définissant ces attributs sont éliminées de la spécification. Après cette transformation, on obtient une grammaire attribuée sur la syntaxe de départ où seuls apparaissent les attributs nécessaires à notre problème particulier. Il est possible de réutiliser cette grammaire attribuée telle quelle ou en la complétant avec de nouveaux attributs et de nouvelles règles sémantiques.

7.2.3 Simplification de la syntaxe

Dorénavant nous supposons que nous ne voulons plus garder la syntaxe de départ. Nous cherchons à trouver une syntaxe minimale pour notre problème précis qui permettra de faciliter l'écriture de grammaire attribuée de passage vers cette grammaire.

Dans la grammaire attribuée résultante des opérations précédentes il est possible d'éliminer deux types de productions :

- les productions qui ne portent aucun calcul. Elles sont appelées productions mortes.
- les productions qui étaient utiles au problème général mais qui maintenant ne servent plus qu'au transport d'attributs par des règles de copie (cf. figure 7.3). Elles sont appellées productions de passage.

Certaines productions peuvent également être simplifiées sans être éliminées, en supprimant certaines occurrences de non-terminaux, appelées occurrences de non-terminaux morts.

Nous remarquons qu'ici nous avons affaire au problème inverse de l'instanciation d'un gène sur une syntaxe. Les productions mortes font partie de branches mortes, les autres correspondent aux productions où la sémantique est constituée de règles de copie.

Élimination des parties mortes

Soit une production $p : X_0 \rightarrow X_1 \ldots X_i \ldots X_n$. Il est possible d'éliminer X_i de la partie droite de cette production dans les cas suivants :

1. X_i est un terminal et il n'est argument d'aucune règle sémantique de la production ;

2. X_i est un non-terminal et il n'y pas d'attributs sur ce non-terminal.

Cette transformation est effectuée sur toutes les productions où cela est possible.

Nous obtenons des productions d'arité nulle où le non-terminal en partie gauche ne porte aucun attribut. Ces productions sont éliminées de la grammaire attribuée.

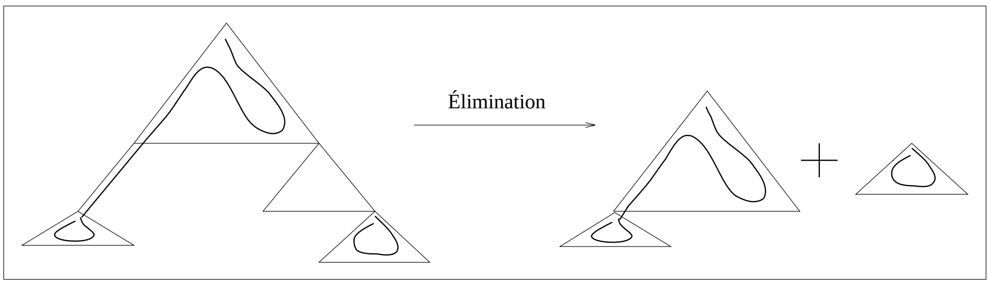


Figure 7.3: Simplification de la syntaxe d'une grammaire attribuée

Élimination des productions de passage

Une fois la transformation précédente effectuée, les productions qui portent les propagations d'informations à travers l'arbre sont des productions unaires (cf. figure 7.4).

Définition 40 (Production de Passage) *On appelle production de passage, toute production $p : X_0 \rightarrow X_1$ telle que toutes les règles sémantiques sont des règles de copie entre une occurrence d'attribut de X_0 et l'occurrence d'attribut de même nom sur X_1. Il peut en outre y avoir sur les non-terminaux une occurrence de l'attribut de sortie ou des attributs d'entrée.*

Il est possible d'éliminer une production de transport dans deux cas :

- X_0 et X_1 sont les occurrences d'un même non-terminal.

- on peut remplacer X_0 par X_1, noté $\mathcal{X}_0 > \mathcal{X}_1$, ou le contraire, dans toutes les productions sans perdre en "généralité". On retombe après remplacement du non-terminal dans le cas précédent.

Afin de pouvoir définir plus précisément la relation $>$ entre les non-terminaux dans une grammaire attribuée, on introduit la notion de graphe de passage.

Définition 41 (Graphe de Passage) *On appelle graphe de passage d'une grammaire attribuée, le graphe orienté sur l'ensemble N des non-terminaux de la grammaire, noté $(N, \rightarrow)$ auquel on a ajouté la possibilité de marquer un sommet. Il existe un arc, $\mathcal{X} \rightarrow \mathcal{Y}$, si et seulement s'il existe une production de passage $p : X \rightarrow Y$. Un non-terminal $\mathcal{X}$ est marqué si et seulement s'il existe une production qui ne soit pas de passage dont il est le père.*

On peut préciser maintenant cette notion de " peut-être remplacé sans perdre en généralité".

Définition 42 ($>$) *Un non-terminal $\mathcal{X}_0$ est $>$ à un autre non-terminal $\mathcal{X}_1$ si dans le graphe de passage de la grammaire attribuée, quel que soit le chemin partant de $\mathcal{X}_0$ on passe par $\mathcal{X}_1$ avant de passer par un non-terminal marqué.*

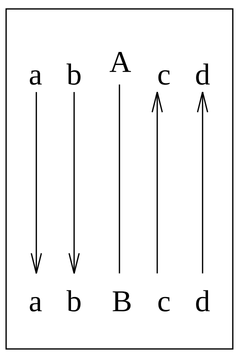


Figure 7.4: Exemple de production de passage

L'algorithme suivant nous permet de trouver si deux non-terminaux $\mathcal{X}$ et $\mathcal{Y}$ vérifient cette propriété :

Algorithme vérifiant si $\mathcal{X} > \mathcal{Y}$
Éliminer les arcs entrants dans $\mathcal{Y}$;
Tant que $\mathcal{X}$ a des arcs sortants **faire**
  Vérifier que tous les voisins en sortant de $\mathcal{X}$ ne sont pas marqués ;
  Si cette propriété est vraie **alors**
    Écraser tous les voisins en sortant de $\mathcal{X}$ sur celui-ci ;
  Sinon

Stop "$\mathcal{X}$ n'est pas $>$ à $\mathcal{Y}$" ;
Fin si
Fin tant que
Écrire "$\mathcal{X} > \mathcal{Y}$" ;

Pour déterminer l'ensemble des relations entre non-terminaux il faut faire ce test pour l'ensemble des couples possibles. Nous avons donc une complexité théorique en n^3 sur le nombre de non-terminaux de la grammaire attribuée.

Il est facile de démontrer que l'on génère ainsi un treillis entre les non-terminaux de la grammaire attribuée. On en déduit un ordre de remplacement entre les non-terminaux de la grammaire attribuée. Avant chaque remplacement, on effectue l'union des attributs pour conserver un éventuel attribut d'entrée ou de sortie.

Création d'une racine

En général, la grammaire attribuée ainsi obtennue n'a plus de racine. D'autre part, si l'on désire que l'attribut résultat de la grammaire attribuée soit remonté à la racine (on veut une grammaire fonctionnelle), il est nécessaire de générer les règles sémantiques qui le remontent.

Supposons le cas général où l'attribut résultat est l'unique attribut de la racine. On a l'algorithme suivant :

1. On crée si elle n'existe pas déjà une racine $\mathcal{Z}$.
2. On associe à la racine un attribut z tel que $type(z)$ est égal au type de l'attribut qui contient le résultat b, s'il n'existe pas déjà ;
3. Pour chaque production $p : X_0 \rightarrow X_1 \ldots X_n$
   - Si $\mathcal{X}_0$ ne porte aucun attribut alors on remplace ce dernier par $\mathcal{Z}$;
   - Si $\mathcal{X}_0 = \mathcal{B}$ alors on génère la production $\mathcal{Z} \rightarrow \mathcal{B}$.
4. Si b est un attribut hérité :
   - on crée sur $\mathcal{B}$ un attribut synthétisé, r de même type que b ;
   - on repère les productions où le non-terminal $\mathcal{B}$ est en partie gauche et on génère une règle de copie, $B_0.r = B_0.b$.
5. Répéter jusqu'à la convergence
   - Si sur une production $p : X_0 \rightarrow X_1 \cdots X_i \cdots X_n$ il existe une occurrence d'attribut $X_i.a$ avec $a = b$ ou $a = r$ qui n'est pas utilisée, alors créer sur $\mathcal{X}_0$ l'attribut r, si celui-ci n'existe pas déjà et rajouter la règle de copie $X_0.r = X_i.a$.
6. Si sur certaines productions on obtient une double définition de r cela veut dire que l'on a remonté plusieurs valeurs de b, on demande alors à l'utilisateur de choisir entre ces valeurs.

7. S'il existe des productions où r n'est pas défini, on demande à l'utilisateur donner une définition compatible avec le type r.

On a ainsi généré une grammaire attribuée fonctionnelle complète. Elle peut servir de gène dans une nouvelle application.

Remarque générale

Il est clair que la généricité de la grammaire ainsi obtenu dépend beaucoup de la généricité de la syntaxe de la grammaire de départ. Il faut donc bien être conscient de ses besoins avant de simplifier une syntaxe. Quoi qu'il en soit ce résultat peut très bien être le départ d'une spécification de gène, à la main.

Il paraît également intéressant de remarquer qu'il est possible de faire des analogies entre l'extraction de gène et l'instanciation. Là où il y élimination dans l'extraction il y a construction dans l'instanciation.

Chapitre 8

Conclusion

Dans ce mémoire, nous avons présenté un ensemble d'algorithmes et de résultats théoriques concernant les grammaires attribuées. Notre but était de proposer des algorithmes de base permettant de rendre la spécification par grammaires attribuées encore plus attrayante. En effet, de nombreux travaux se sont intéressés à l'augmentation de la puissance d'expression des grammaires attribuées et à l'amélioration de l'efficacité des évaluateurs produits ; il est maintenant possible d'utiliser des systèmes de traitements de grammaires attribuées, tels que FNC-2, qui tiennent compte des nombreux résultats qui en ont découlés. Dans le même temps, très peu de travaux ont été consacrés à l'introduction de concepts déjà anciens, comme la modularité ou la réutilisabilité, que l'on retrouve dans la plupart des langages de programmation.

Le point de départ de cette étude a donc été d'essayer d'inclure ces concepts au sein du système FNC-2, tout d'abord en essayant d'adapter les notions de grammaires attribuées fonctionnelles et de méta-composition introduites par Ganzinger et Giegerich. Les limitations de l'algorithme de base ainsi que la "mauvaise" qualité des grammaires attribuées produites, nous ont amenés à orienter nos recherches dans deux directions : la première a cherché à augmenter le pouvoir d'expression de la méta-composition et la deuxième à transformer les grammaires attribuées produites afin d'améliorer la qualité des évaluateurs qui leurs sont associés. Nous avons obtenu des résultats intéressants dans les deux cas et ils ont été implantés au sein du système FNC-2 dans le Coupleur.

Malheureusement, le peu d'exemples de grammaires attribuées écrites sous forme de grammaires attribuées fonctionnelles ne nous ont pas permis de tester l'efficacité du Coupleur dans le cadre de l'écriture d'une application complète. De plus, l'étude du problème nous a montré quelques limitations de la méta-composition qui, dans le cadre du système FNC-2, nous ont paru être des obstacles à une l'utilisation systématique de la méta-composition. La non-clôture de la classe FNC vis-à-vis de la méta-composition était un premier obstacle qu'il ne nous a pas été possible de contourner. Nous ne sommes pas capables de déduire de la spécification de deux grammaires attribuées fonctionnelles si leur composition sera FNC ou non, sans effectuer un test d'une complexité comparable à la méta-composition suivie du test FNC. Cette construction peut donc se révéler coûteuse dans certains cas. D'autre part, d'un point de vue purement pratique la méta-composition est coûteuse en temps de construction car elle ne permet pas une compilation séparée.

Ces deux remarques montrent que l'outil méta-composition est peu adapté à la mise au point d'une application. En revanche, son utilisation en fin de processus d'implantation permet, compte tenu des améliorations que nous avons apportées, de construire un évaluateur qui ne souffre pas de l'écriture modulaire (construction intermédiaire d'arbres) et qui donc sera plus efficace en temps et en place mémoire.

Le "savoir faire" acquis lors de l'étude de la méta-composition ainsi que les différentes remarques faites précédemment nous ont amenés à introduire une nouvelle vision de la composition. Nous voulions que celle-ci conserve les bonnes propriétés de la méta-composition (pas de construction des arbres intermédiaires), qu'elle ne soit pas limitée au niveau de la clôture des classes et qu'elle permette une compilation séparée dans la mesure du possible. C'est pourquoi nous avons introduit la composition des évaluateurs, qui est une variante de la méta-composition qui, au lieu de travailler au niveau des spécifications, travaille au niveau des évaluateurs de grammaires attribuées. Cette notion remplit bien les deux premières conditions que nous nous étions fixés, mais la compilation séparée n'est pas encore atteinte. Il a fallu pousser un peu plus loin la composition des évaluateurs et introduire la notion de filtre de couplage, qui permet effectivement une compilation séparée. Si la composition simple des évaluateurs semble ne pas être coûteuse au niveau de l'efficacité des évaluateurs produits par rapport à un évaluateur créé après la méta-composition, cette conclusion est beaucoup moins évidente lorsque l'on utilise les filtres de couplage. Il est donc nécessaire d'effectuer une implantation de ces filtres pour se faire une idée exacte. Le travail déjà effectué sur le système FNC-2 pour l'implantation du Coupleur nous permet de penser que cette implantation ne devrait pas poser de problème majeur.

Les différents résultats concernant l'élargissement de la classe des grammaires attribuées acceptées ainsi que les différentes optimisations apportées à la construction des évaluateurs nous permettent, de plus, d'envisager la composition des évaluateurs comme un outil performant autant au niveau de la mise au point d'une application, qu'au niveau de la fin du processus d'implantation.

La modularité introduite par les grammaires attribuées fonctionnelles ne semble pas toujours adaptée à l'écriture d'une application qui peut se séparer en différents sous-problèmes interdépendants sur une même syntaxe. C'est la raison pour laquelle nous avons introduit une écriture modulaire complémentaire, avec les grammaires attribuées incomplètes. Celles-ci permettent de décomposer une grosse application en différentes grammaires attribuées incomplètes, toutes décrites sur une même syntaxe et qui, grâce à des attributs d'entrée et de sortie, communiquent les unes avec les autres. La méta-union, comme la méta-composition, permet de construire une grammaire attribuée ayant la même sémantique qu'un ensemble de modules. La méta-union a également les mêmes limitations que la méta-composition et, même si quelques résultats peuvent nous permettre de conclure quant à la clôture de classe à moindre coût, il paraît coûteux d'utiliser la méta-union dans une phase de mise au point. C'est la raison pour laquelle nous avons introduit la liaison des évaluateurs qui permet, grâce à une liaison dynamique entre les évaluateurs des différents modules, une véritable compilation séparée. Même si nous décrivons différentes constructions pour ces évaluateurs, il semble que cette décomposition puisse se révéler coûteuse au niveau de la "synchronisation" des évaluateurs mais aussi au niveau du recalcul de certains attributs. Là encore, seule une implantation et une comparaison des différentes méthodes pour des cas pratiques pourra nous dire si, par rapport à la méta-union ces méthodes permettent de produire des évaluateurs efficaces. Toutefois, il est à noter que ces méthodes peuvent nous permettre de construire des évaluateurs de type DNC là où la méta-union ne le permet pas.

Le second concept que nous voulions introduire dans les grammaires attribuées était la réutilisabilité. Pour cela nous avons choisi l'approche des grammaires attribuées génériques, ou gènes, développée dans [9]. La base de cette approche consiste à dire que, pour pouvoir facilement réutiliser une spécification sous forme de grammaire attribuée, il faut que celle-ci soit faite sur une syntaxe minimale (générique). En effet, il est alors plus simple de construire une grammaire attribuée fonctionnelle faisant passer d'une syntaxe réelle vers la syntaxe de la grammaire attribuée générique. Il est ainsi possible d'instancier la grammaire attribuée générique sur la syntaxe réelle. Nous avons proposé ici un algorithme de base qui permet de construire de façon semi-automatique cette grammaire attribuée fonctionnelle. On trouvera certaines améliorations de l'algorithme dans [9], mais il reste toujours à l'implanter pour vérifier que les hypothèses que nous avons faites sont valables. Nous avons ensuite introduit un algorithme d'extraction de gènes qui, combiné avec l'algorithme précédent, permet une véritable réutilisation de morceaux de spécifications déjà écrites. Nous pensons que cet outil, bien que très simple, peut être très utile au programmeur, même si la grammaire attribuée produite n'atteint pas toujours le "statut" de gène.

Tous ces algorithmes permettent d'augmenter la facilité d'utilisation des grammaires attribuées et nous espérons ainsi rendre la spécification sous forme de grammaire attribuée encore plus attrayante. D'un point de vue théorique, ils ont aussi permis de montrer que tout n'avait pas encore était dit sur la façon d'évaluer les grammaires attribuées et donner une vision un peu différente de la vision classique, notamment quant au rôle de l'arbre.

Nous espérons que les grammaires attribuées arriveront un jour à se libérer des grammaires qui les restreignent un peu, pour s'attaquer, par exemple, aux graphes. En effet, nous avons montré que le rôle joué par la partie droite d'une production était équivalent à celui de la partie gauche dans les méthodes d'évaluation de type DNC. Il est alors possible de voir une production comme un sommet de graphe et les non-terminaux comme les arcs sortant de ce sommet ; on profite ainsi de tous les résultats concernant les grammaires attribuées sur cette nouvelle méthode de spécification, que nous pourrions appeler graphe attribué. Même si les graphes attribués ne font pas une longue carrière, il paraît important que les bienfaits, encore méconnus, des grammaires attribuées puissent être appliqués à d'autres domaines que les compilateurs. Nous espérons que ce travail y contribuera.

Bibliographie

[1] Pierre Deransart, Martin Jourdan & Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. #323, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[2] Donald E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* 2 (June 1968), 127–145, Correction: *Math. Systems Theory* 5, 1, pp. 95-96 (Mar. 1971)..

[3] Kablan Barbar, "Étude comparative de différentes classes de grammaires d'attributs ordonnées," Univ. de Bordeaux I, thèse de 3ème cycle, June 1982.

[4] Harald Ganzinger & Robert Giegerich, "Attribute Coupled Grammars," *ACM SIGPLAN Notices* 19 (June 1984), 157–170.

[5] Rodney Farrow, Thomas J. Marlowe & Daniel M. Yellin, "Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation," in *19th ACM Symp. on Principles of Progr. Languages*, Jan. 1992, 223–234.

[6] Gerald D. P. Dueck & Gordon V. Cormack, "Modular Attribute Grammars," *Comput. J.* 33 (1990), 164–172, See also: research report CS-88-19, Univ. of Waterloo (May 1988).

[7] Claude Kleinhans, "Intégration de la modularité dans les grammaires attribuées," Dépt. d'Informatique, Univ. de Nice, Thèse, Feb. 1991.

[8] Beate Baum, "Another Kind of Modular Attribute Grammars," in *4th Int. Conf. on Compiler Construction*, U. Kastens & P. Pfahler, eds., Lect. Notes in Comp. Sci. #641, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1992, 44–50.

[9] Carole Le Bellec, "La généricité et les grammaires attribuées ," Univ. d'Orléans, Thése , June 1993.

[10] Gregor V. Bochmann, "Semantic Evaluation from Left to Right," *Comm. ACM* 19 (Feb. 1976), 55–62.

[11] Isu Fang, "FOLDS, a Declarative Formal Language Definition System," Comp. Sc. Dept., Stanford Univ., PhD thesis, report STAN-CS-72-329, Dec. 1972, Abstract in: *Séminaires Structure et Programmation des Calculateurs 1973*, ed. M. Kronental and Bernard Lorho, INRIA, Rocquencourt, pp. 275-290 (1973)..

[12] Bernard Lorho, "De la définition à la traduction des langages de programmation: méthode des attributs sémantiques," Univ. Paul Sabatier, thèse d'État, Toulouse, Nov. 1974.

[13] Bernard Lorho, "Semantic Attributes Processing in the System DELTA," in *Methods of Algorithmic Language Implementation*, A. Ershov & Cornelius H. A. Koster., eds., Lect. Notes in Comp. Sci. #47, Springer-Verlag, New York–Heidelberg–Berlin, 1977, 21–40.

[14] R. Cohen & E. Harry, "Automatic Generation of Near-Optimal Linear-Time Translators for Non-Circular Attribute Grammars," in *6th ACM Symp. on Principles of Progr. Languages*, Jan. 1979, 121–134.

[15] Ken Kennedy & J. Ramanathan, "A Deterministic Attribute Grammar Evaluator based on Dynamic Sequencing," *ACM Trans. Progr. Languages and Systems* 1 (July 1979), 142–160.

[16] Ole L. Madsen, "On Defining Semantics by means of Extended Attribute Grammars," in *Semantics-Directed Compiler Generation*, Neil D. Jones, ed., Lect. Notes in Comp. Sci. #94, Springer-Verlag, New York–Heidelberg–Berlin, 1980, 259–299, See also: report DAIMI PB-109, Comp. Sc. Dept., Aarhus Univ. (Jan. 1980)..

[17] Fahimeh Jalili, "A General Linear-Time Evaluator for Attribute Grammars," *ACM SIGPLAN Notices* 18 (Sept. 1983), 35–44.

[18] Martin Jourdan, "Recursive Evaluators for Attribute Grammars: an Implementation," in *Methods and Tools for Compiler Construction*, Bernard Lorho, ed., Cambridge Univ. Press, Cambridge, 1984, 139–164.

[19] Pierre Deransart, Martin Jourdan & Bernard Lorho, "Speeding up Circularity Tests for Attribute Grammars," *Acta Inform.* 21 (Dec. 1984), 375–391.

[20] Martin Jourdan & Didier Parigot, *The FNC-2 System User's Guide and Reference Manual*, INRIA, Rocquencourt, Feb. 1989, This manual is periodically updated.

[21] Martin Jourdan, "Un évaluateur efficace pour les grammaires attribuées fortement non-circulaires," Laboratoire d'Informatique Théorique et Programmation, rapport 82-39, Paris, Sept. 1982.

[22] Martin Jourdan, "Les grammaires attribuées: implantation, applications, optimisations," Univ. Paris VII, Thèse de Docteur-Ingénieur, May 1984.

[23] Martin Jourdan, "Strongly Non-Circular Attribute Grammars and Their Recursive Evaluation," *ACM SIGPLAN Notices* 19 (June 1984), 81–93.

[24] Bruno Courcelle & Paul Franchi-Zannettacci, "Attribute Grammars and Recursive Program Schemes," *Theoret. Comput. Sci.* 17 (1982), 163–191 and 235-257.

[25] Paul Franchi-Zannettacci, "Attributs sémantiques et schémas de programmes," Univ. de Bordeaux I, thèse d'État, Mar. 1982.

[26] Kablan Barbar, "Classification des grammaires d'attributs ordonnées," Univ. de Bordeaux I, Rapport 8412, Apr. 1984.

[27] Gilberto Filé, "Classical and Incremental Attribute Evaluation by Means of Recursive Procedures," in *11th Coll. on Trees in Algebra and Programming (CAAP '86)*, Paul Franchi-Zannettacci, ed., Lect. Notes in Comp. Sci. #214, Springer-Verlag, New York–Heidelberg–Berlin, Mar. 1986, 112–126.

[28] Gilberto Filè, "Classical and Incremental Attribute Evaluation by Means of Recursive Procedures," *Theoret. Comput. Sci.* 53 (Jan. 1987), 25–65.

[29] Pierre Deransart, "Validation des grammaires d'attributs," Univ. de Bordeaux I, thèse d'État, Oct. 1984.

[30] Joost Engelfriet, "Attribute Grammars: Attribute Evaluation Methods," in *Methods and Tools for Compiler Construction*, Bernard Lorho, ed., Cambridge Univ. Press, Cambridge, 1984, 103–138.

[31] Uwe Kastens, "Ordered Attribute Grammars," *Acta Inform.* 13 (1980), 229–256.

[32] Joost Engelfriet & Gilberto Filé, "Simple Multi-Visit Attribute Grammars," *J. Comput. System Sci.* 24 (June 1982), 283–314.

[33] Gilberto Filé, "Interpretation and Reduction of Attribute Grammars," *Acta Inform.* 19 (1983), 115–150.

[34] Didier Parigot, "Mise en œuvre des grammaires attribuées: transformation, évaluation incrémentale, optimisations," Univ. de Paris-Sud, Thèse de 3ème cycle, Orsay, Sept. 1987.

[35] Martin Jourdan & Didier Parigot, "Techniques for Improving Grammar Flow Analysis," in *ESOP '90*, May 1990.

[36] Takehiro Tokuda, "Two Methods for Eliminating Redundant Copy Operations from the Evaluation of Attribute Grammars," *J. Inform. Process.* 9 (1986), 79–85.

[37] Kari-Jouko Räihä, "A Space Management Technique for Multi-pass Attribute Evaluators," Dept. of Comp. Sc., Univ. of Helsinki, PhD thesis, report A-1981-4, Sept. 1981.

[38] Pierre Boullier, Philippe Deschamp & Bernard Lorho, *The SYNTAX Reference Manual*, INRIA, Rocquencourt, Nov. 1987.

[39] Robert Giegerich, "Composition and Evaluation of Attribute Coupled Grammars," *Acta Inform.* 25 (1988), 355–423.

[40] Bruno Marmol, Univ. d'Orléans, thèse d'État, Non encore soutenue.

[41] Martin Jourdan, Carole Le Bellec, Didier Parigot & Gilles Roussel, "Specification and Implementation of Grammar Couplings using Attribute Grammars," in *Programming Languages Implementation and Logic Programming*, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1993.