Rapport de stage

effectué à l'INRIA-Rocquencourt dans le projet

CHLOÉ

TRANSFORMATION DE GRAMMAIRES ATTRIBUEES POUR DES MISES A JOUR DESTRUCTIVES

Etienne DURIS

D.E.A. d'Informatique - Université d'Orléans

Septembre 1994

Remerciements

Je remercie tout d'abord Martin Jourdan pour m'avoir proposé ce stage et pour l'avoir encadré. Je sais qu'il avait depuis longtemps l'idée de ce sujet et je suis ravi qu'il m'ait choisi pour l'étudier. Il a su me faire profiter de son expérience dans le domaine des grammaires attribuées.

Je remercie également Didier Parigot à qui ce travail doit beaucoup. Il m'a conseillé dans mes recherches et malgré ses occupations, il a toujours trouvé le temps de répondre à mes nombreuses questions.

Je tiens aussi à remercier l'ensemble du projet ChLoÉ pour son accueil et sa disponibilité, sans oublier les thésards et les stagiaires. Merci à Gilles Roussel pour ses explications et l'intérêt qu'il a porté à mon travail. Merci à Bruno Marmol pour ses nombreux conseils et sa bonne humeur. Merci à Florence Bridon pour son humour et ses relectures laborieuses et à Sylvain Lelait pour ses dégustations...

Table des matières

Chapitre 1

Introduction

L'évolution des langages de programmation du langage binaire vers des langages de plus haut niveau tels que PL/1, Algol et jusqu'à Prolog, Ada ou CAML procure au programmeur un pouvoir d'expression et un confort de plus en plus grand. Dans le même temps, les performances "brutes" des machines augmentent aux dépens de leur simplicité. Le lien sémantique entre les langages de programmation et les architectures des machines est donc un problème fondamental en informatique et la conséquence immédiate de cet écart est la complexité grandissante des compilateurs.

Parmi les différentes méthodes de "description" des langages de programmation, certaines s'intéressent aux aspects sémantiques qui dépendent du contexte : on dit qu'elles traitent de la sémantique statique. C'est le cas des doubles grammaires, des grammaires affixes ou des **grammaires attribuées**.

Les grammaires attribuées sont une méthode formelle de spécification des contraintes d'un langage de programmation permettant de construire automatiquement des compilateurs. Depuis qu'elles sont nées il y a plus de vingt ans, la construction d'évaluateurs très efficaces en temps de calcul et en espace mémoire a donné lieu à de nombreux travaux de recherche.

Le système de traitement des grammaires attribuées FNC-2 en cours de développement dans le projet CHLOÉ de l'INRIA-Rocquencourt est un des exemples les plus achevés de ces travaux.

Les auteurs de grammaires attribuées, en particulier les utilisateurs d'FNC-2 et de son langage OLGA, ont mis au point un certain nombre de types de données abstraits dont l'implantation n'effectue pas d'effet de bord (conformément aux exigences des grammaires attribuées). Par exemple le type `symbol-table` est muni de la fonction suivante :

```
function insert (st : symbol-table ; n : nom ; i : info) : symbol-table
```

qui insère l'entrée (n,i) dans la table `st`, à ceci près que son premier argument `st` n'est pas modifié, mais une nouvelle table des symboles est construite et rendue. On appelle ces fonctions dont le résultat est "semblable" à l'un de leurs arguments des **fonctions de mise à jour**.

Le partage de structure permet en général de diminuer la consommation de mémoire, mais l'obligation d'applicativité entraîne une certaine perte d'efficacité en place et en temps. Par exemple, si la table de symboles est implantée comme un arbre binaire de recherche, la nouvelle table ne diffère de l'ancienne que par le chemin de la racine jusqu'au point d'insertion, dont tous les nœuds doivent être dupliqués (obligation d'applicativité). Cependant, tous les autres sous-arbres sont partagés (ils appartiennent à la fois à `st` et au résultat de la fonction `insert`).

Par ailleurs, le système FNC-2 effectue sur les grammaires attribuées qu'il traite des analyses poussées visant à mieux gérer la mémoire. Au cours de ces analyses, il peut quelquefois détecter qu'un appel à une fonction de mise à jour constitue la dernière utilisation d'un argument. Dans le cas de notre exemple, si un appel à `insert` constitue de manière prévisible la dernière utilisation de `st`, l'évaluateur pourrait être rendu plus efficace si, au lieu de dupliquer tous les noeuds du chemin, il se contentait de modifier l'argument `st` au point d'insertion et de le retourner comme résultat de la fonction.

Dans ce rapport, nous étudions la possibilité de construire automatiquement la version **destructive** correspondant à une fonction de mise à jour applicative donnée. Par "destructive", on entend le fait qu'elle puisse modifier physiquement la structure de son argument mis à jour, évitant ainsi de nombreuses allocations mémoire. D'autre part, nous devons déterminer sous quelles conditions un appel à une fonction de mise à jour applicative dans une grammaire attribuée peut être remplacé par un appel à sa version destructive.

Le chapitre 2 rappelle succinctement les définitions et les notations utilisées dans la théorie des grammaires attribuées. Le chapitre 3 donne un aperçu du fonctionnement du système FNC-2 et décrit en particulier la représentation des grammaires attribuées qu'il accepte en entrée : c'est le langage OLGA.

A l'aide de ces notions, nous présentons plus précisément notre problème dans le chapitre 4. Celui-ci contient un bref état de l'art dans les domaines qui nous concernent et nous y justifions notre choix de traiter ce problème dans le cadre des grammaires attribuées plutôt que dans un cadre purement fonctionnel.

Le chapitre 5 concerne la construction automatique de la version destructive d'une fonction de mise à jour. Etant donné le cadre choisi pour notre étude, nous représentons une fonction de mise à jour par une grammaire attribuée et nous fournissons un algorithme de transformation qui, à une grammaire attribuée et un évaluateur donnés, associe un évaluateur destructif qui a la même sémantique.

Nous discutons dans le chapitre 6 des conditions dans lesquelles un appel à une fonction de mise à jour dans une grammaire attribuée peut être remplacé par un appel à sa version destructive. Ce remplacement peut être décidé en fonction d'une propriété de destructibilité dont nous exhibons un algorithme de calcul.

Nous montrons comment ces optimisations peuvent être mises en place au sein du système FNC-2 actuel dans le chapitre 7 et les annexes fournissent des exemples d'applications.

Les résultats que nous décrivons dans ce rapport permettent d'optimiser les évaluateurs produits par Fnc-2 en réutilisant autant que possible des structures complexes au lieu de les dupliquer. Dans ce domaine, nous avons été très intéressés par les travaux de Philip Wadler sur la déforestation [Wa 88] et ceux de Gilles Roussel sur la méta-composition [Ro 94]. En effet, ces deux mécanismes ont pour but d'éviter la construction de structures intermédiaires, le premier dans le cadre des langages fonctionnels et le second dans le cadre des grammaires attribuées. Nous présentons dans le chapitre 9 une comparaison de ces deux mécanismes dans leurs effets et sur leurs domaines d'action respectifs. Ce dernier chapitre n'est qu'un résumé des observations que nous avons pu faire, mais il peut constituer une amorce de rapprochement entre la programmation purement fonctionnelle et les grammaires attribuées.

Chapitre 2

Définitions et notations

2.1 Grammaires attribuées

Dans ce chapitre, nous donnons un certain nombre de définitions et de notations pour décrire de façon formelle les grammaires attribuées. La thèse de Gilles Roussel [Ro 94], qui traite d'un sujet assez proche du nôtre, constitue dans ce domaine une référence détaillée et complète. En reprenant son travail et ses notations, nous nous contenterons ici de donner le formalisme nécessaire à la compréhension et à l'utilisation des notions que nous manipulerons tout au long de ce rapport.

Définition 1 (Grammaire indépendante du contexte)
On appelle grammaire indépendante du contexte (CF-grammaire) un quadruplet G=(N,T,P,Z) avec :

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

Nous supposons que l'analyse du langage *L(G)* engendré par la grammaire G permet de construire les arbres syntaxiques correspondants.

Nous supposons aussi que lors de l'analyse d'une phrase, il est possible de récupérer la valeur des terminaux, conformément à leur type sémantique associé.[1]

1. Comme dans [Ro 94], nous verrons les non-terminaux comme des **types syntaxiques** et les terminaux comme des **types sémantiques**. Par contre, pour alléger les notations, nous confondrons les terminaux (resp. non-terminaux) qui représentent les types avec les occurrences de terminaux (resp. occurrences de non-terminaux) qui interviennent effectivement dans les productions.

A chaque production de P de la forme :

$$(X_0,X_1,\ldots,X_n) \text{ où } n \geq 0 \text{ , } X_0 \in N \text{ et } X_i \in N \cup T \text{ , } 1 \leq i \leq n$$

est associée une production nommée de la forme :

$$p : X_0 \to X_1 \ldots X_n$$

p est le nom unique de la production, et par abus de langage nous confondrons une production avec son nom ou avec sa production nommée associée.

Définition 2 (Grammaire attribuée)
Une grammaire attribuée est un triplet AG=(G,A,F) avec :

- **G**=*(N,T,P,Z) une grammaire indépendante du contexte,*
- **A** *l'ensemble des attributs des non-terminaux ;*

$$A = \bigcup_{X \in N} A(X)$$

où **A(X)***, l'ensemble des* **attributs** *de X est la réunion disjointe de* **S(X)***, l'ensemble des attributs* **synthétisés** *de X et de* **H(X)***, l'ensemble des attributs* **hérités** *de X. Si* **a** *est un attribut de A(X), il est également noté* **X.a***.*

Pour une production $p : X_0 \to X_1 \ldots X_n$ donnée, sont aussi définis les ensembles suivants :

  - *l'ensemble des* **occurrences d'attributs** *de p :*

$$W(p) = \{X_i.a \text{ tels que } 0 \leq i \leq n \text{ , } a \in A(X_i)\}$$

  - *l'ensemble des* **occurrences d'attributs définies** *de p :*

$$DO(p) = \{X_i.a \text{ tels que } (i = 0 \ \wedge \ a \in S(X_0)) \ \vee \ (1 \leq i \leq n \ \wedge \ a \in H(X_i))\}$$

  - *et l'ensemble des* **occurrences d'attributs utilisées**

$$UO(p) = W(p) - DO(p)$$

- **F** *un ensemble de règles sémantiques associées aux productions ;*

$$F = \bigcup_{p \in P} E_p$$

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

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

avec $X_{f(i)}.a_i \in UO(p)$*, f une fonction de* $\{1,\ldots,q\}$ *dans* $\{0,\ldots,n\}$ *et* $F_{p,a,k}$ *une fonction*[2]*.*

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 non circulaire qui n'est pas en forme normale, d'obtenir une grammaire attribuée équivalente en forme normale [Bo 76].

2.2 Arbre d'entrée

Une grammaire attribuée est une méthode déclarative de spécification d'un calcul à effectuer sur un objet structuré, que l'on appelle l'arbre d'entrée. Ce dernier est en fait l'arbre correspondant à l'analyse de la "phrase" d'entrée. Ses feuilles portent les valeurs qui seront utilisées pour effectuer le calcul.

Définition 3 (Arbre d'entrée) *pour une grammaire attribuée.*

Un arbre d'entrée t pour une grammaire attribuée AG=(G,A,F) est un arbre syntaxique associé à une phrase du langage L(G).

Un nœud de l'arbre t est noté u et un nœud feuille de celui-ci est noté l. A chaque nœud u est associée une production $label(u) = p : X_0 \rightarrow X_1 \cdots X_n$ *telle que :*

- *u est associé au non-terminal* X_0 *(comme* $X_0 \in N$*, u est de type syntaxique) ;*
- *u a n fils (u est d'arité n) ;*
- *si* u_i *est le nœud fils de u en position i, alors* u_i *est associé au non-terminal* X_i *(il est de type syntaxique) ;*
- *si* l_j *est le nœud feuille fils de u en position j, alors* l_j *est associé au terminal* X_i *(il est de type sémantique).*

Nous dirons que : $u \rightarrow \ldots l_j \ldots u_i \ldots$ *est une instance de la production* $label(u)$*.*

Remarquons dès à présent qu'un calcul ne peut se faire que sur un seul arbre d'entrée à la fois : les valeurs nécessaires aux calculs sont obtenues par l'analyse et le parcours de sa structure.

2. cette fonction est en fait l'abstraction d'un terme sur une signature $\Sigma = (\mathcal{K}, \mathcal{F})$ pour $UO(p)$ où $T \subset \mathcal{K}$ et $N \cap \mathcal{K} = \emptyset$.

2.3 Evaluation

Etant donnée 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 aux règles sémantiques de la grammaire attribuée.

Plus précisément :

– à chaque nœud u de l'arbre d'entrée t est associé un ensemble d'**instances d'attributs** :

$$IA(u) = \{u.a \ tel \ que \ a \in A(X_0)\}$$

où $label(u) = p : X_0 \rightarrow X_1 \cdots X_n$.

– à chaque nœud feuille l de t, est associé le singleton : $IA(l) = \{l.\epsilon\}$ où 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 :

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

est associée 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 la valeur de l'instance d'attribut synthétisé z de la racine. Le fait qu'il n'y ait qu'un seul attribut synthétisé à la racine n'est pas une restriction gênante, puisque ajouter une règle sémantique qui encapsule ces différents résultats au niveau de la racine ne change pas les dépendances entre les occurrences d'attributs. Le problème revient donc à calculer uniquement les valeurs des instances d'attributs qui sont nécessaires au calcul de z.

Le système $K(t)$ a une unique solution s'il n'est pas circulaire. Nous dirons 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.

Les méthodes d'évaluation de la valeur sémantique se basent sur la notion de **graphe de dépendance**, qui fournit l'ordre d'évaluation des instances d'attributs entre eux.

Définition 4 (Graphe de dépendance local)
A chaque production de la forme $p : X_0 \rightarrow X_1 \ldots X_n$ *est associé un graphe de dépendance local* $D(p)$ *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 si il existe une règle sémantique de la forme :*

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

Cette dépendance est notée $X_j.b \xrightarrow{p} X_i.a$.

Par abus de langage, nous confondons ce graphe de dépendance local avec la relation qui lui est associée $\xrightarrow{p}$. Etant donnée une relation R (ou son graphe), R^+ dénote la fermeture transitive de R.

Par extension, le **graphe de dépendance global** sur un arbre t est défini comme l'union des instanciations des graphes de dépendance locaux sur l'ensemble des instances de productions de t. Sa relation associée est notée $\xrightarrow{t}$.

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 de $\xrightarrow{t}^{+}$. Cette méthode très simple a été utilisée dans de nombreux systèmes. 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 a une complexité exponentielle en fonction de la taille de la grammaire [DJL 84].

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 pour le calcul des instances d'attributs. Les classes les plus importantes sont les classes L et Sweep. La classe L permet de calculer tous les attributs en parcourant l'arbre en plusieurs passes de gauche à droite. La classe Sweep revient à la classe L moyennant une permutation de la partie droite de certaines productions. Un certain nombre de systèmes sont basés sur cette approche [DJL 88]. Les évaluateurs de cette famille sont très efficaces, mais la classe des grammaires attribuées acceptées est très restreinte.

Une troisième famille d'évaluateurs est basée sur des méthodes dites statiques. Celles essayent, à partir des graphes de dépendance locaux, de déduire statiquement un ordre d'évaluation des occurrences d'attributs d'une production qui soit valable quel que 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 structure de la grammaire attribuée qui induit l'ordre. La classe des grammaires attribuées acceptées est donc plus large.

Notons que ce stage se déroule au sein du système FNC-2 [JP 89] qui produit des évaluateurs basés sur une telle méthode statique acceptant des grammaires attribuées *l-ordonnées*. Ces évaluateurs à séquences de visites sont totalement déterministes. [En 84] [Ka 80].

Chapitre 3

Le système Fnc-2

3.1 Présentation générale

Fnc-2 est un système de traitement de grammaires attribuées développé à l'INRIA [JP 89]. Le travail que nous avons effectué se situe dans le cadre de Fnc-2 dont nous allons donner une présentation simple. Le but ici n'est pas de détailler le fonctionnement du système, mais plutôt de décrire les différentes parties et de fournir les notions importantes qui seront utilisées tout au long de ce rapport.

Le système Fnc-2 accepte en entrée la spécification d'une grammaire attribuée à partir de laquelle il produit un évaluateur [1]. Cet évaluateur est en fait un code exécutable qui, pour un arbre d'entrée, calcule la valeur sémantique spécifiée par la grammaire attribuée.

La figure 3.1 présente le schéma général du système. Il accepte en entrée la spécification d'une grammaire attribuée, grâce aux langages Olga et Asx [2]. La partie frontale du système (le *Front End*) analyse et contrôle la syntaxe de la spécification d'entrée. Elle fournit au générateur d'évaluateurs les informations structurelles concernant la grammaire attribuée.

Le générateur d'évaluateurs effectue des tests de caractérisation de la grammaire attribuée pour connaître sa classe et éventuellement la transformer en une grammaire l-ordonnée. Il génère ensuite des séquences de visites correspondant à l'ordre d'évaluation des attributs et effectue des optimisations mémoire. Ce générateur produit un évaluateur abstrait, en ce sens qu'il ne contient ni les règles sémantiques, ni les autres fonctions spécifiées dans la grammaire attribuée d'entrée. Il y fait simplement référence. De plus, il est indépendant de tout langage d'implantation.

Toutes ces règles sémantiques et ces appels à des fonctions sont mis à part par le front end, et envoyés à des traducteurs (c'est le *Back End*). Le rôle du back end est double :

- d'une part il transcrit les règles sémantiques et les appels de fonctions dans les différents langages cibles prévus,

1. L'évaluateur associé à une grammaire attribuée n'est pas nécessairement unique. En général, il y a plusieurs façon d'obtenir le résultat dont elle spécifie le calcul.

2. La section suivante décrit ces langages.

– d'autre part il prend en entrée l'évaluateur abstrait et produit les codes correspondants (dans les différents langages cibles) en y incorporant les codes des règles sémantiques.

Nous obtenons donc en sortie du back end des évaluateurs "concrets", qui sont des codes exécutables. Un tel évaluateur prend en entrée un arbre et effectue sur celui-ci le calcul spécifié par la grammaire attribuée d'entrée.

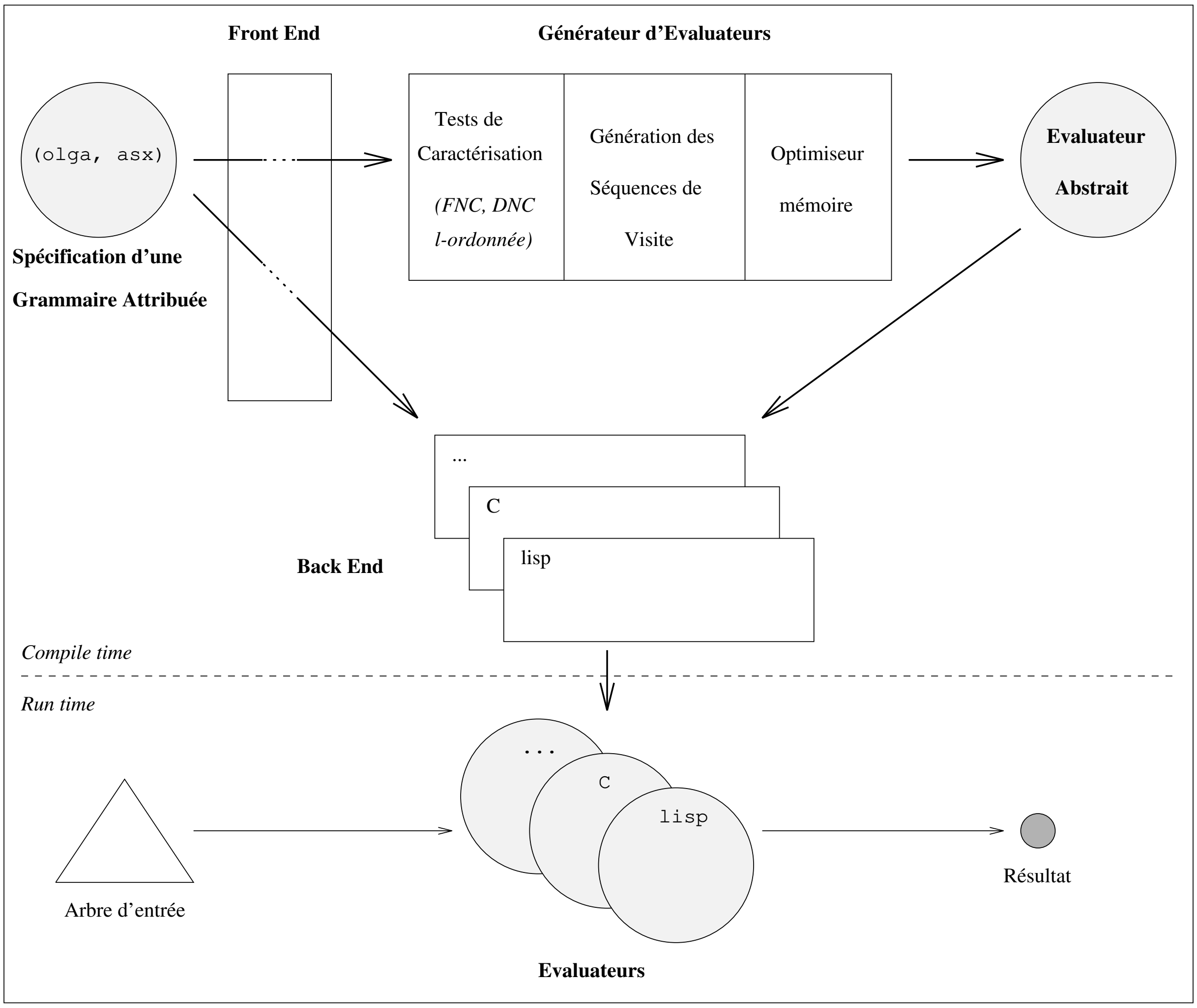


FIG. 3.1 - *Le système* FNC-2.

3.2 Le langage OLGA

Pour décrire ces grammaires attribuées, le système FNC-2 dispose d'un **langage applicatif fortement typé**. Il s'agit du langage OLGA. Nous considérerons ici que le langage OLGA accepte les extensions et les syntaxes nouvelles qui sont en cours de développement actuellement dans le projet CHLOÉ. En effet, l'esprit dans lequel nous travaillons est celui du langage NEWOLGA qui n'est pas encore implanté. Les extensions autorisées dans NEWOLGA sont principalement :

- la possibilité d'avoir des attributs hérités à la racine ;
- la prise en compte des non-terminaux des productions en tant que variables que l'on peut utiliser dans les règles sémantiques.

Dans la suite, nous parlerons donc d'OLGA même si certaines propriétés n'y sont pas encore implantées.

Syntaxe abstraite : le langage ASX

OLGA contient un sous-langage, ASX, qui permet de définir les syntaxes abstraites. Ce sont ces syntaxes abstraites qui décrivent les structures syntaxiques (ou arbres abstraits) sur lesquelles un évaluateur produit par FNC-2 va travailler.

Pour une grammaire attribuée $AG = (G, A, F)$, un programme ASX définit des **phyla** et des **opérateurs**, associés respectivement aux non-terminaux et aux productions de G ; il caractérise ainsi les traits du langage $L(G)$.

Les opérateurs correspondent aux nœuds d'un arbre d'entrée. Ils peuvent être d'arité fixe ou variable. Un opérateur d'arité fixe et nulle correspond à une feuille. Il représente un terminal de la grammaire non-contextuelle G. Les opérateurs d'arité fixe et non-nulle peuvent avoir des descendants de différentes sortes, tandis que ceux qui sont d'arité variable doivent avoir tous leurs descendants de la même sorte. Ces sortes sont formalisées par la notion de **phylum**.

Les phyla sont des ensembles non vides et disjoints d'opérateurs. A chaque position de descendance d'un opérateur est associé un phylum. Celui-ci contient contient les opérateurs pouvant engendrer des sous-arbres à cette position[3].

Pour définir une syntaxe abstraite, on doit donc décrire les opérateurs et les phyla, ce qui constitue un programme ASX[4]. Il faut définir chaque opérateur avec son profil (i.e. son arité et les phyla qui sont associés à ses fils) et chaque phylum avec les opérateurs qu'il contient.

Nous pouvons par exemple représenter la structure d'arbre binaire d'entiers par le programme ASX de la figure 3.2.

`AXIOM-TREE` et `TREE` sont des phyla et `axiom-tree`, `fork` et `tip` sont des opérateurs. `int` est le type entier prédéfini dans OLGA.

3. Un même phylum peut être associé à différentes positions de descendance de différentes productions.
4. La définition complète de la syntaxe ASX est donnée dans [JP 89].

```
grammar AXIOM-TREE is

     root is axiom-tree ;

     AXIOM-TREE = axiom-tree ;
     axiom-tree -> TREE ;

     TREE = fork tip ;
     fork -> TREE TREE ;
     tip -> int ;

end grammar ;
```

FIG. 3.2 - *Programme* ASX *définissant la structure d'arbre binaire d'entiers.*

Grammaire attribuée

Un évaluateur généré par le système FNC-2 ne manipule pas uniquement des structures syntaxiques. Il faut, en plus du programme ASX, définir les attributs qui décorent les arbres abstraits. Le tout constitue la syntaxe abstraite attribuée décrivant les arbres décorés qui sont manipulés par un évaluateur.

Supposons que l'on ait un programme ASX qui définisse une syntaxe abstraite avec ses opérateurs et ses phyla. Alors un programme OLGA permet de définir une grammaire attribuée en spécifiant :

- les attributs qui complètent la syntaxe abstraite en syntaxe abstraite attribuée,
- pour chaque production p, les règles sémantiques qui spécifient les valeurs des occurrences d'attributs définies (de $DO(p)$).

L'en-tête d'un programme OLGA donne le profil de la grammaire attribuée. Ce profil spécifie la syntaxe abstraite qu'elle prend en entrée, les types (phyla) des arguments sémantiques ainsi que la syntaxe abstraite du résultat.

Les attributs sont définis en spécifiant pour chacun d'entre eux son nom (commençant par un \$), sa classe (hérité ou synthétisé), le phylum auquel il est attaché et son type.

Une occurrence d'attribut synthétisé est définie à un nœud n d'un arbre en fonction des valeurs de ses occurrences aux nœuds fils de n : pour cette raison on dit que son sens de propagation dans l'arbre syntaxique est du bas vers le haut (on dit aussi que sa valeur "remonte").

Par opposition, un attribut hérité se propage du haut vers le bas ; sa valeur "descend" l'arbre. La valeur de son occurrence à un nœud dépend de la valeur de son occurrence au nœud père de n.

Le reste d'un programme OLGA spécifie pour chaque règle de production p de la grammaire G les règles sémantiques définissant les éléments de $DO(p)$. Les règles de productions sont décrites par la syntaxe abstraite : un opérateur a pour descendants plusieurs phyla. La syntaxe OLGA est la suivante :

```
where    operateur  →  P1 ... Pn    use
      ...
      (définition des attributs de DO(p) par les règles sémantiques)
      ...
end where ;
```

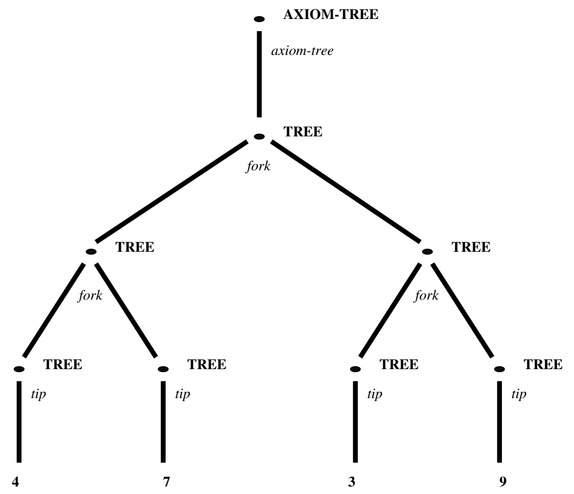
où $P_1 \ldots P_n$ sont des noms de variables qui représentent les phyla dans lesquels l'opérateur dérive.

Par exemple, le programme de la figure 3.3 permet de construire un arbre binaire d'entiers "minimum" (cf. figure 3.2), ayant la même structure que l'arbre binaire d'entrée. Il est minimum en ce sens que la valeur de toutes ses feuilles est la plus petite des valeurs des feuilles de l'arbre d'entrée.

L'attribut synthétisé `$min` remonte la plus petite valeur des feuilles. Une fois cette valeur propagée par l'attribut hérité `$rmin` c'est l'attribut synthétisé `$tree` qui remonte la valeur de l'arbre en cours de construction jusqu'à `$axiom-tree` (attribut de la racine). Ce dernier est la *valeur sémantique* de la grammaire attribuée, l'attribut synthétisé à la racine : c'est le résultat de la grammaire attribuée. La figure 3.4 illustre ce calcul.

```
attribute grammar min-tree(AXIOM-TREE) : AXIOM-TREE is
attribute
     synthesized $min(TREE) : int ;
     synthesized $tree(TREE) : TREE ;
     synthesized $axiom-tree(AXIOM-TREE)  : AXIOM-TREE ;
     inherited $rmin(TREE) : int ;

where axiom-tree -> TREE use
     $rmin(TREE) := $min(TREE) ;
     $axiom-tree := axiom($tree(TREE)) ;
end where ;

where fork -> TREE-1 TREE-2 use
     $min := if ($min(TREE-1) ≤ $min(TREE-2)) then
                  $min(TREE-1)
             else
                  $min(TREE-2)
             end if ;
     $rmin(TREE-1) := $rmin(fork) ;
     $rmin(TREE-2) := $rmin(fork) ;
     $tree := fork($tree(TREE-1),$tree(TREE-2)) ;
end where ;

where tip -> VAL use
     $min := VAL ;
     $tree := tip($rmin(tip)) ;
end where ;

end grammar ;
```

FIG. 3.3 - *Programme* OLGA *construisant l'arbre minimum.*



Un exemple d'arbre binaire d'entier.

Ci-contre, l'arbre d'entrée avec les noms des phyla et des opérateurs ainsi que les valeurs aux feuilles.

Ci-dessous, représentation des dépendances et de la construction du résultat.

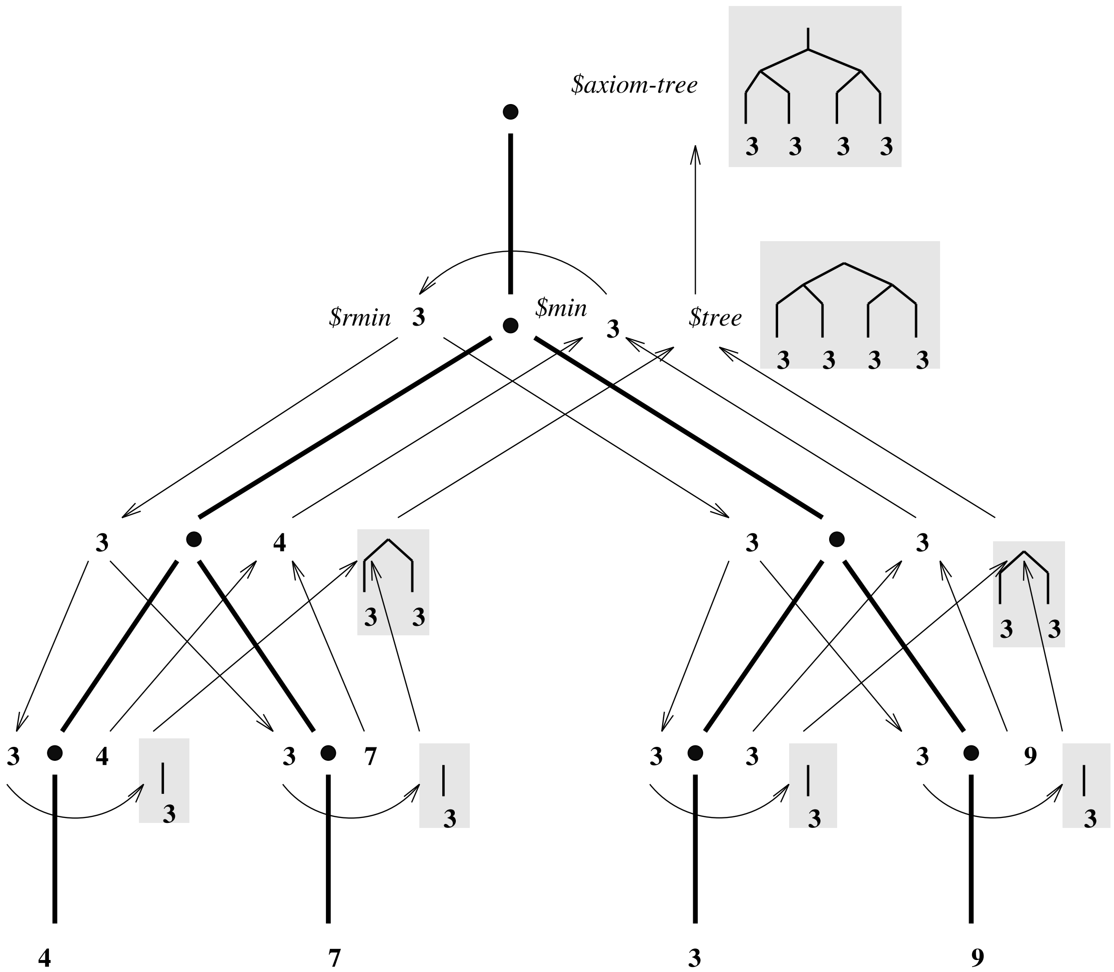


FIG. 3.4 - *Calcul de l'arbre minimum.*

3.3 Gestion mémoire

L'évaluation de la valeur sémantique d'un arbre t passe par le calcul d'un nombre important d'attributs. Le générateur d'évaluateurs de FNC-2 effectue diverses optimisations mémoire. La principale consiste à essayer d'implanter plusieurs instances d'attributs dans la même variable et nécessite de connaître la *durée de vie* des attributs (cf. définition ci-dessous). Cette information est calculée statiquement par des algorithmes de FNC-2 [Ju 86][Pa 88].

Définition 5 (Durée de Vie)
La durée de vie d'une instance d'attribut est l'intervalle de temps entre le moment de sa création, son calcul proprement dit, et le moment de sa mort (sa dernière utilisation). La durée de vie d'une instance détermine l'espace de temps d'occupation de son espace physique.

Pour illustrer brièvement cette notion, regardons les deux exemples suivants. Chacun représente un graphe de dépendance local sur une même production. Dans les deux cas, les attributs sont les mêmes, mais les règles sémantiques sont différentes. L'ordre d'évaluation peut être le même dans les deux cas. Par exemple : (X.h, Y1.h, Y1.s, Y2.h, Y2.s, X.s). Dans

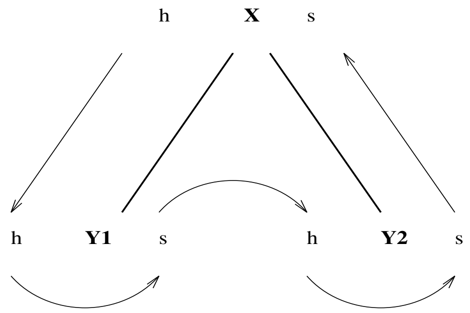


FIG. 3.5 - *Durée de vie courte de Y1.s.*

l'exemple de la figure 3.5 la durée de vie de l'attribut Y1.s, va de son calcul en fonction de Y1.h jusqu'à son utilisation pour le calcul de Y2.h. Après et avant, cet attribut est *mort.*

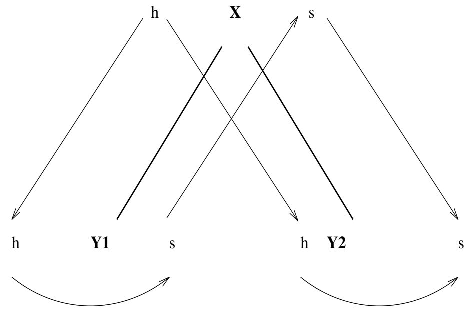


FIG. 3.6 - *Durée de vie longue de Y1.s.*

Dans la figure 3.6 la durée de vie de Y1.s, qui commence à sa définition en fonction de Y1.h, ne se termine qu'au calcul de X.s.

Chapitre 4

Présentation du problème

4.1 Le sujet

Dans les grammaires attribuées, l'interdiction de faire des effets de bords dans les règles sémantiques (due au non-déterminisme de la résolution du système $K(t)$) offre de nombreux avantages : sémantique plus claire, possibilités d'optimisations en temps et en place, etc. Les auteurs de grammaires attribuées, en particulier les utilisateurs du système FNC-2, utilisent des types de données abstraits structurés et des fonctions de mise à jour sur ces objets. Ces fonctions doivent être applicatives pour éviter les effets de bord et, pour cette raison, elles sont écrites dans le langage applicatif OLGA. Ce sont des fonctions de mise à jour dans le sens qu'elles retournent un résultat "semblable" à l'un de leurs arguments (de même type).

Il se trouve que fréquemment, l'argument de ce type de fonction n'est plus utilisé (en tant que tel) après l'appel à la fonction. Au sens de sa durée de vie, il est "mort". De façon naïve, on pourrait réutiliser la structure de l'argument pour construire le résultat de la fonction. Ceci éviterait un nombre important d'allocations mémoire (pour le résultat) et de désallocations (pour l'argument). Ce procédé est aussi quelquefois appelé *update in place* dans les langages fonctionnels.

Cependant, ces fonctions étant applicatives, elles engendrent le partage de certaines structures.

Définition 6 (Partage mémoire)
Les fonctions applicatives permettent (pour gagner de la place) à des objets complexes de partager entre eux les mêmes espaces physiques. Des objets différents peuvent être construits à partir de sous-unités communes (éléments de listes ou autres structures chaînées) et ils peuvent donc partager physiquement ces dernières.

Pour illustrer la notion de partage mémoire, nous donnons l'exemple de la mise à jour d'une table des symboles représentée par un arbre binaire de recherche (cf. figure 4.1). La fonction *insert-in-tree* prend en entrée un arbre binaire de recherche dans lequel elle doit insérer un nouvel élément en fonction d'une clé.

Cette fonction applicative va donc construire l'arbre résultant en créant un nœud pour la valeur à insérer ; elle va dupliquer les nœuds du chemin qui mène au point d'insertion et le reste de la structure de l'arbre d'entrée sera "partagé" par le résultat.

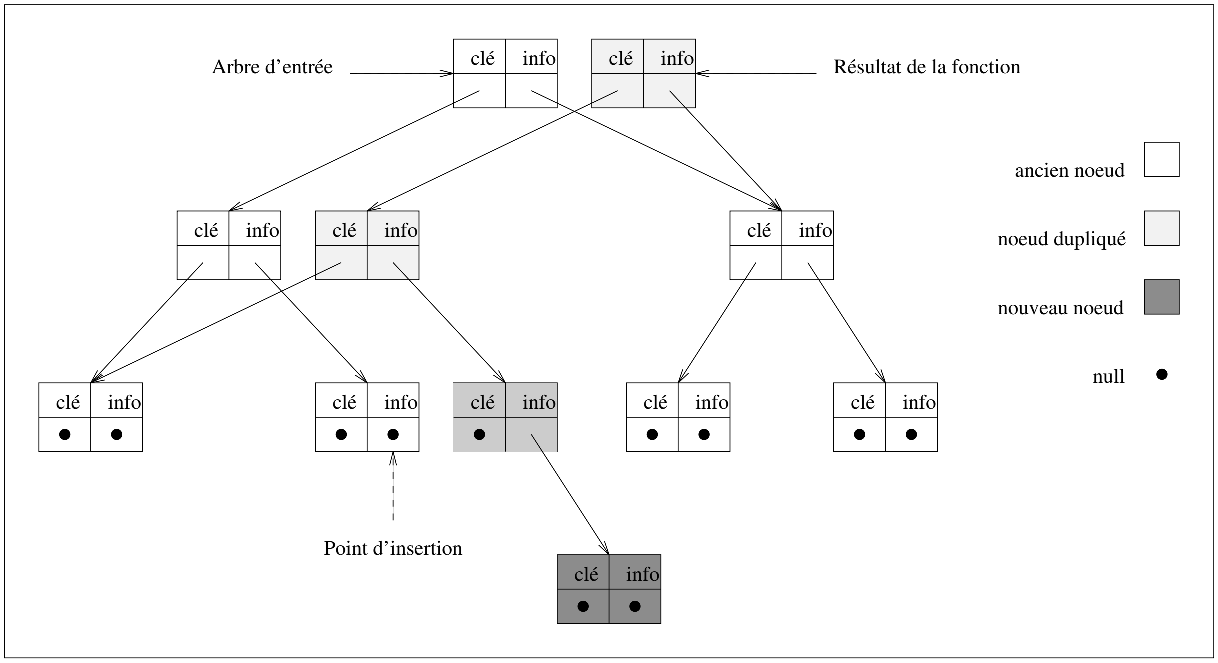


FIG. 4.1 - *Arbres partagés par la fonction applicative insert-in-tree.*

Ce que nous voudrions faire, sur cet exemple particulier, c'est construire automatiquement (à partir de la fonction *insert-in-tree*) une fonction qui n'allouerait de la place mémoire que pour le nouveau nœud à insérer et qui réutiliserait les autres noeuds (ceux du chemin de la racine au point d'insertion) au lieu de les dupliquer. Nous appellerons ce genre de fonction une fonction destructive, puisqu'elle modifie physiquement l'arbre d'entrée pour construire le résultat.

Définition 7 (Fonction destructive)
Soit f une fonction applicative qui prend n arguments de types $T_1 \ldots T_n$ et qui retourne un résultat de type T_i. Son profil est donc le suivant :

$$f\ :T_1,\ldots,T_n\ \rightarrow\ T_i$$

On appelle fonction destructive associée à f la fonction ayant le même profil, mais dont le résultat est construit (en totalité ou en partie) en effectuant des modifications physiques sur les arguments d'entrée.

La création automatique de telles fonctions est un problème de transformation de programme en fonction de la structure du résultat construit, mais son utilisation et sa mise en place n'est pas possible systématiquement. Puisque l'argument d'entrée est modifié physiquement, il faut s'assurer qu'il n'est jamais utilisé après l'appel à la fonction destructive et qu'il n'est pas partagé par d'autres variables encore en vie, sous peine de fausser le résultat attendu.

Nous voyons donc que le problème de la création automatique de fonctions destructives et de leur mise en place requiert des informations sur la structure des résultats construits, sur la durée de vie des arguments et sur leur partage.

Les travaux existants dans ce domaine

Nous ne sommes pas les premiers à avoir besoin de la notion de partage des variables d'un programme. Dans les langages de programmation fonctionnelle, le même problème se pose dès qu'il est question de récupération de mémoire au moment de la compilation[1]. L'avènement de l'**interprétation abstraite** à la suite des travaux de Patrick et Radhia Cousot [CC 77] a donné lieu à beaucoup de recherches. Parmi celles-ci, citons les plus importantes concernant la récupération mémoire au moment de la compilation dans les langages fonctionnels :

- Paul Hudak [Hu 86] a travaillé sur un modèle sémantique de **comptage de référence**. Une interprétation abstraite de ce modèle sur un programme permet de connaître le nombre de références à un ensemble de structures de données. Si un tel ensemble n'est référencé qu'une seule fois, alors sa mise à jour peut être faite de manière destructive plutôt que par recopie des structures.

- James Larus et Paul Hilfinger [LH 88] ont décrit la notion d'**"alias graph"** qui permet de comparer des objets référencés à différents points d'un programme pour détecter des conflits potentiels entre ces objets.

- Lucy Hederman a fait en 1988 un état de l'art assez complet des utilisations du comptage de référence pour la récupération mémoire au moment de la compilation [He 88].

- Le concept d'**analyse de chemins** a également été étudié en tant que "trace" de l'évolution possible d'un objet dans un programme. Les événements pouvant provoquer un partage sont pris en compte par ces chemins.

  Jones et Le Métayer [JLM 89] ont utilisé l'interprétation abstraite pour projeter un domaine de chemins infini sur un domaine fini afin d'étudier les conflits de structures dans un programme.

  Sestoft a introduit dans [Se 89] les concepts de *chemin définition-utilisation*, *chemin sémantique*, *interférence de chemins* et *grammaire de définition-utilisation* qui dérive un sur-ensemble des chemins possibles pour un programme.

  Cette notion de chemin d'analyse est aussi utilisée par Adrienne Bloss [Bl 89] avec une interprétation abstraite pour décider d'effectuer des mises à jour destructives.

- Alain Deutsch a présenté dans [De 90] une méthode d'analyse statique pour l'estimation du partage et de la **durée de vie** de données allouées dynamiquement, utilisant un modèle opérationnel des programmes fonctionnels sur lequel il construit une interprétation abstraite.

1. En anglais : compile-time garbage collection.

Il a continué ses travaux sur les problèmes d'interférences et de partage [De 92]. Ses dernières recherches sur la détection inter-procédurale du partage utilisent la notion de **paire de chemins d'accès symbolique** et procurent une information nettement plus précise que les travaux précédents quant à la structure des objets partagés.

Cette liste n'est pas exhaustive, mais elle montre que les méthodes de détection du partage et des durées de vie dans les langages fonctionnels utilisent pratiquement toutes la notion d'interprétation abstraite. Les résultats obtenus sont des approximations plus ou moins précises de l'information que l'on recherche et mettent en œuvre des mécanismes complexes et coûteux.

Le contexte d'étude étant le système FNC-2, nous avons aussi considéré que les grammaires attribuées pouvaient être un outil adéquat pour traiter nos problèmes. Les travaux de Didier Parigot [Pa 88] sur l'évaluation et l'optimisation des grammaires attribuées ont permis de mettre en place dans le système FNC-2 des algorithmes efficaces calculant **de façon exacte la durée de vie** des attributs [JP 89].

Dans les domaines purement fonctionnels, la durée de vie d'une variable est une information délicate à calculer et le plus souvent approchée (interprétation abstraite). L'avantage des grammaires attribuées est donc très net dans ce domaine.

D'autre part, la spécification **guidée par la syntaxe** d'un calcul, l'accès aux productions (construisant le résultat) et le typage fort que fournit le langage OLGA facilitent la transformation de programme. Par exemple, il nous sera facile de situer dans du code OLGA chaque étape de la construction d'un résultat.

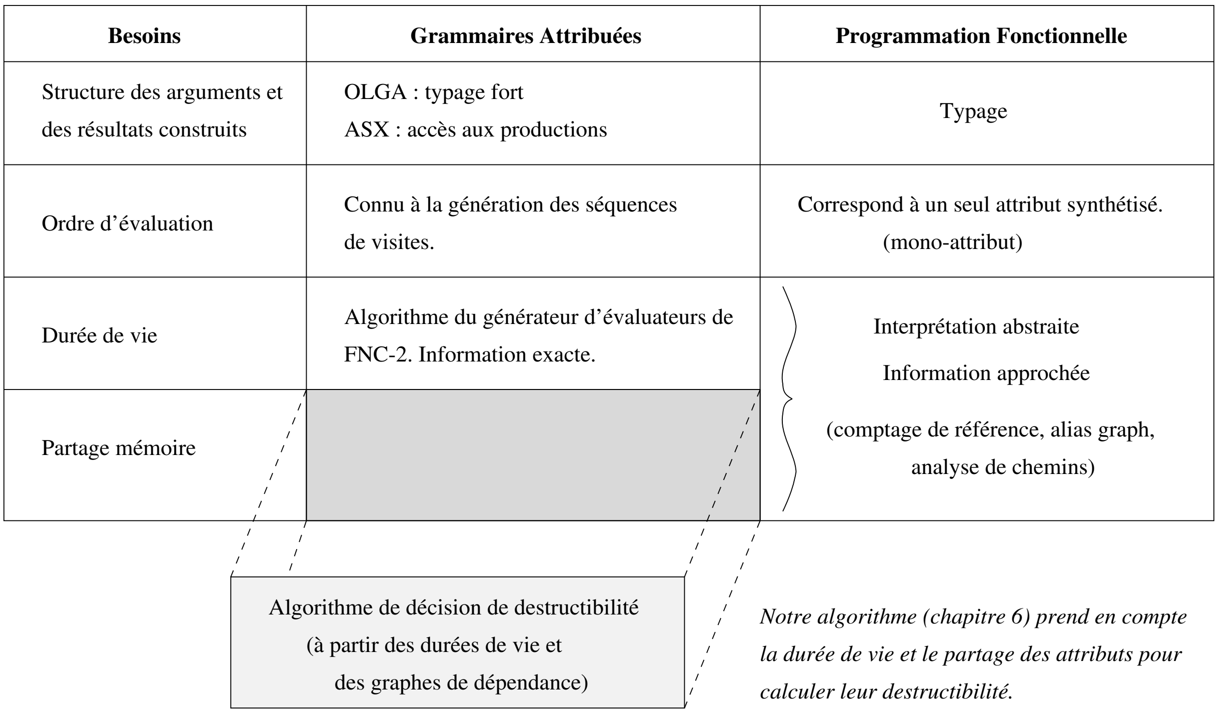


FIG. 4.2 - *Les outils disponibles pour traiter notre problème.*

Le tableau de la figure 4.2 présente les différentes informations dont nous avons besoin ainsi que les outils et les méthodes permettant d'obtenir ces information dans le domaine purement fonctionnel et dans le cadre des grammaires attribuées de FNC-2. Nous présentons au chapitre 6 un algorithme qui complète les outils dont nous avons besoin et qui fait des grammaires attribuées un contexte propice à la résolution de notre problème.

Ces observations, ainsi que le contexte de travail que fournit FNC-2, nous ont conduit à limiter notre problème aux grammaires attribuées acceptées par notre système.

4.2 Le problème ramené aux grammaires attribuées

Le contexte dans lequel nous étudions la transformation de fonctions applicatives en fonctions destructives est celui d'FNC-2. L'application globale est alors une grammaire attribuée dont les règles sémantiques font appel à des fonctions applicatives elles aussi écrites en OLGA.

Dans ce cas, nous pouvons considérer qu'une fonction écrite en OLGA est une grammaire attribuée, dont l'attribut synthétisé à la racine[2] est le résultat et dont l'arbre d'entrée est l'argument principal. Les autres arguments de la fonction sont alors des attributs hérités à la racine (que l'on peut voir comme des initialisations d'attributs sémantiques)[3].

Dans ces conditions, nous disposons d'informations précises sur les arguments (les attributs) quant à leur durée de vie, leurs dépendances entre eux et leur partage.

Une fonction OLGA ainsi représentée possède alors un évaluateur associé qui, à un arbre d'entrée, associe le résultat de cette fonction. Si nous construisons une version destructive de cet évaluateur, elle pourra éventuellement être mise en place à chaque occurrence d'appel de la fonction correspondante. Cela veut dire qu'à une fonction donnée, on associe un évaluateur destructif une fois pour toutes. Dans l'application globale, on décidera pour chaque appel à cette fonction applicative si on appelle son évaluateur “classique” ou l'évaluateur destructif correspondant (cf. figure 4.3).

Le problème se décompose donc en deux grandes parties :

- Comment construire automatiquement un évaluateur destructif à partir d'une grammaire attribuée correspondant à une fonction applicative ? C'est ce que nous appellerons **la transformation**. Nous utiliserons pour cela le formalisme OLGA et l'ordre d'évaluation calculé par FNC-2.

- Sous quelles conditions le résultat de cette transformation peut-il être mis en place dans le contexte de la grammaire attribuée globale qui effectue des appels à ces fonctions applicatives ? C'est un problème d'analyse de flot de données, de durée de vie et de partage de mémoire. Nous appellerons cette étape l'étude de **“destructibilité”**[4].

2. On pourra considérer que plusieurs résultats sont retournés, comme en ML par exemple, où une fonction peut retourner des paires. Ces cas correspondent à des grammaires attribuées ayant plusieurs attributs synthétisés à la racine.

3. C'est l'esprit du langage NEWOLGA en cours de développement, mais pas encore implanté.

4. Le sens de destructible sera explicité dans la section 6.1.

Nous étudierons la transformation de la fonction applicative indépendamment de son contexte d'appel. Elle dépend de la grammaire attribuée représentant cette fonction et de son évaluateur.

D'autre part, la décision du remplacement ou non d'une fonction applicative par sa version destructive est facilitée par les analyses effectuées par le système FNC-2 et par la distinction sémantique/syntaxique (rendue possible par le contexte des grammaires attribuées et par le typage fort d'OLGA). Les règles de productions qui guident les calculs dans un programme OLGA manipulent certains types (phyla). Seuls les attributs de ces types peuvent être analysés dans leur structure par un tel programme. C'est ainsi que l'on peut déterminer qu'un attribut est syntaxique. Inversement, un attribut sémantique est considéré comme une valeur (sa structure est transparente pour l'application).

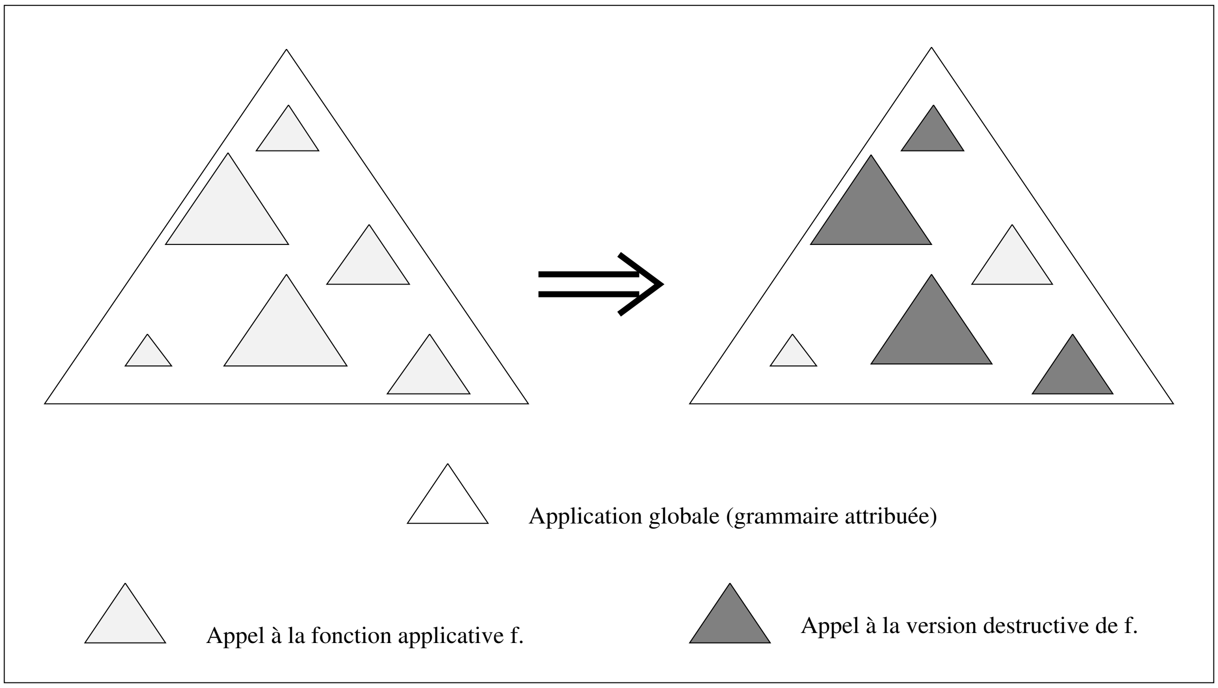


FIG. 4.3 - *La mise en place de la version destructive se décide pour chaque occurrence d'appel.*

Chapitre 5

La transformation

Nous allons présenter dans cette section le problème de la transformation d'une fonction applicative en une fonction destructive. Puisque nous avons choisi notre contexte d'étude, nous considérerons désormais qu'une fonction applicative est décrite par une grammaire attribuée.

5.1 Les restrictions

Nous disposons d'une grammaire attribuée écrite en OLGA qui représente la fonction applicative à rendre destructive. FNC-2 est le système d'évaluation de cette grammaire attribuée[1].

Les fonctions applicatives qui nous intéressent acceptent plusieurs arguments, et retournent un résultat. De façon plus générale, on peut aussi considérer des fonctions qui retournent plusieurs résultats (comme en ML par exemple où l'on peut retourner des paires). Elles ont le profil suivant :

$$f : T_1, \ldots, T_n \;\rightarrow\; (T_{i_1}, \ldots, T_{i_p})$$

Un des arguments de la fonction correspond à l'arbre d'entrée de la grammaire attribuée. Le système FNC-2 et le langage OLGA autorisent la présence d'attributs hérités à la racine d'un arbre d'entrée. Ce sont ces attributs qui véhiculent les informations et les valeurs représentées par les autres arguments de la fonction applicative.

La grammaire attribuée correspondant à cette fonction acceptera alors un arbre d'entrée avec $n-1$ attributs hérités permettant de transmettre les autres arguments. Les p résultats seront représentés par p attributs synthétisés.

On peut noter que les attributs hérités à la racine sont des attributs sémantiques : en aucun cas ils ne seront analysés dans leur structure (le programme OLGA n'est guidé que par la structure de l'arbre d'entrée). Ils sont considérés comme des valeurs. Par opposition, l'arbre

1. Nous ne rentrerons pas dans les détails de FNC-2 et d'OLGA mais nous nous en servirons plutôt comme cadre d'étude et comme moyen de représentation.

d'entrée est syntaxique : le programme Olga va être guidé par l'analyse de sa structure grâce aux `where ... use`.

L'avantage principal de cette restriction (Fnc-2, Olga) est de disposer d'un formalisme syntaxique permettant d'analyser précisément un objet (l'arbre d'entrée) et de pouvoir, au fil de la construction du résultat, en réutiliser des parties.

Le mapping syntaxique

Dans le cas des fonctions de mise à jour, le résultat de la fonction est construit selon la même structure que l'arbre d'entrée. Pour un nœud donné, l'attribut synthétisé (représentant le résultat) est souvent défini avec le même opérateur Olga que celui de la production courante.

Reprenons l'exemple de la structure d'arbre binaire d'entiers (figure 3.2). Le phylum `TREE` est associé à deux opérateurs :

- `fork` construit une structure arborescente vers deux phyla `TREE`,
- `tip` construit un accès à un type simple (entier).

Les noeuds de phylum `TREE` peuvent donc, selon les cas, avoir des structures physiques différentes.

Imaginons qu'un programme Olga analyse un nœud u de l'arbre d'entrée par un

`where fork -> TREE TREE use [...] end where ;`

et que pour cette règle de production un attribut $\$a$ de phylum `TREE` soit défini.

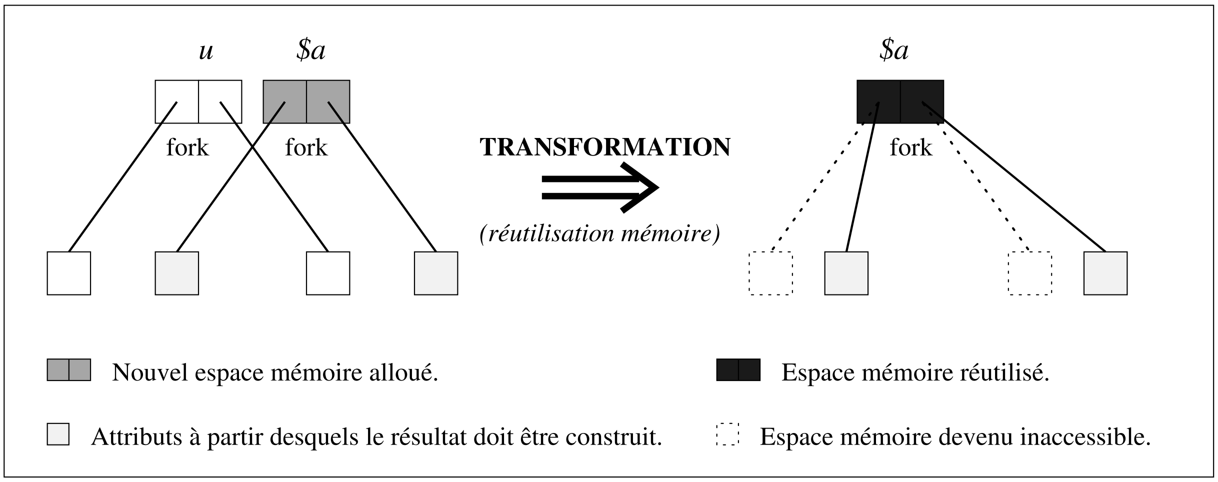


Fig. 5.1 - *Exemple avec mapping syntaxique.*

Si $\$a$ est défini par l'opérateur `fork`, sa structure sera exactement la même (physiquement) que celle de u (voir figure 5.1). Dans ce cas, nous pouvons réutiliser l'espace physique alloué pour u plutôt que de le "dupliquer". Pour cela, il faut modifier le nœud analysé u :

- on affecte aux fils de u les nouvelles valeurs,

– on affecte à $\$a$ la valeur de u, sans aucune allocation mémoire.

Cette réutilisation simple de la place allouée pour un nœud nécessite ce que nous appellerons **le mapping syntaxique** entre la structure du nœud analysé et la structure du nœud construit. On exige en effet que l'opérateur de la règle de production courante soit le même que celui qui construit l'attribut calculé.

Par contre, si $\$a$ est défini par l'opérateur `tip`, il n'aura pas la même structure physique que le nœud analysé (voir figure 5.2). Dans ce cas où il n'y a pas de mapping syntaxique, nous n'effectuerons pas de réutilisation de l'espace mémoire. De façon statique, il est plus délicat de réutiliser de l'espace mémoire sous la forme d'une structure différente[2]. C'est peut être plus le travail d'un ramasse-miettes dynamique (garbage collector).

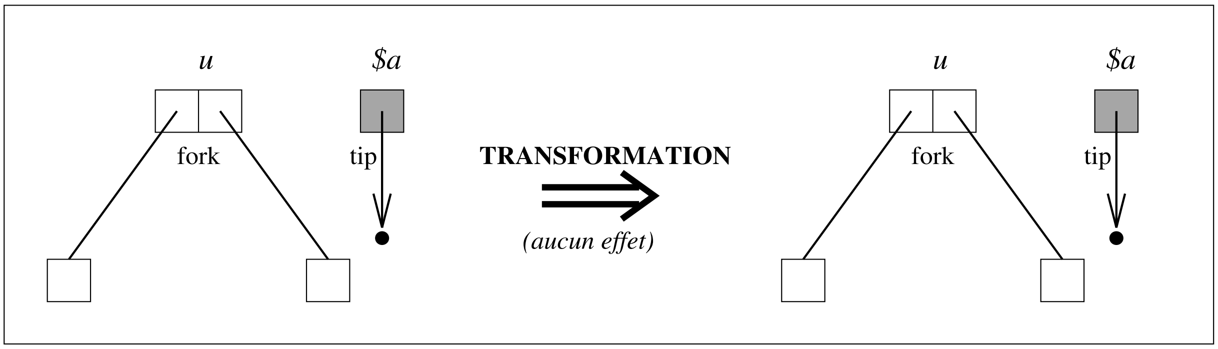


FIG. 5.2 - *Exemple sans mapping syntaxique.*

Du point de vue du code OLGA, la réutilisation d'espace mémoire se fera donc lorsque un attribut `$attr` est construit avec **le même opérateur** que celui de la production courante. Soit, de façon schématique :

```
where cons ->...
      ...
      $attr := cons(...) ;
      ...
end where ;
```

La nécessité de l'ordre d'évaluation

Le principe de base de notre transformation étant de réutiliser de la place mémoire en modifiant les nœuds de l'arbre d'entrée, cela introduit immédiatement une contrainte : il ne faut pas que cette modification gène la suite du calcul.

Nous voyons donc que la notion d'**ordre d'évaluation** est sous-jacente à ce problème. En effet, si le calcul d'un attribut modifie le nœud de l'arbre où il est évalué, il est clair que toutes les informations accessibles à partir de ce nœud sont ensuite invalides. En particulier, il ne faut plus évaluer aucun attribut sur ce nœud modifié.

2. On pourrait par contre désallouer explicitement le nœud courant de l'arbre d'entrée.

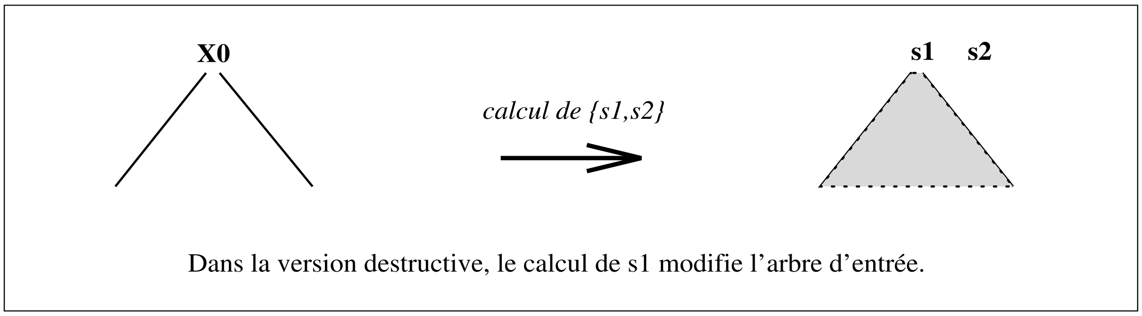


FIG. 5.3 - *Importance de l'ordre d'évaluation des attributs.*

La figure 5.3 présente un exemple où deux attributs (s_1 et s_2) sont évalués sur un même nœud d'un arbre d'entrée. Nous supposons qu'il y a mapping syntaxique entre s_1 et le nœud courant et que cet attribut doit être construit en réutilisant la place mémoire utilisée par le nœud X_0. s_2 est un attribut de type simple. Le résultat que l'on souhaite obtenir pour l'évaluation "destructive" de ces attributs sur le nœud courant provoque donc une modification de l'arbre d'entrée.

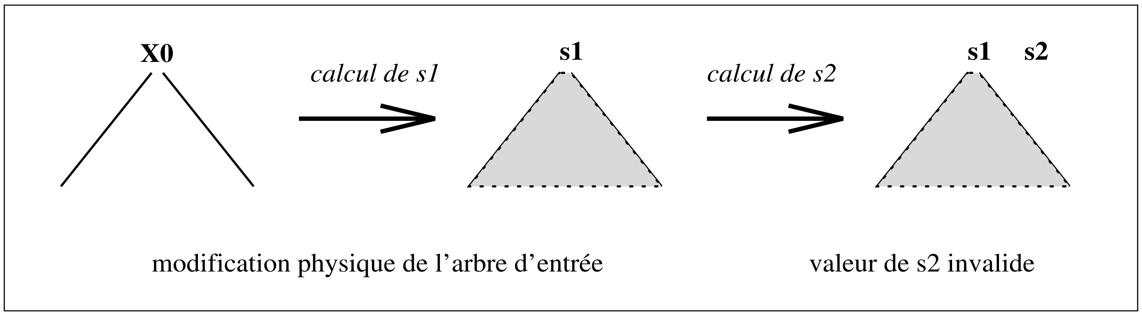


FIG. 5.4 - *Ordre d'évaluation incompatible avec la transformation.*

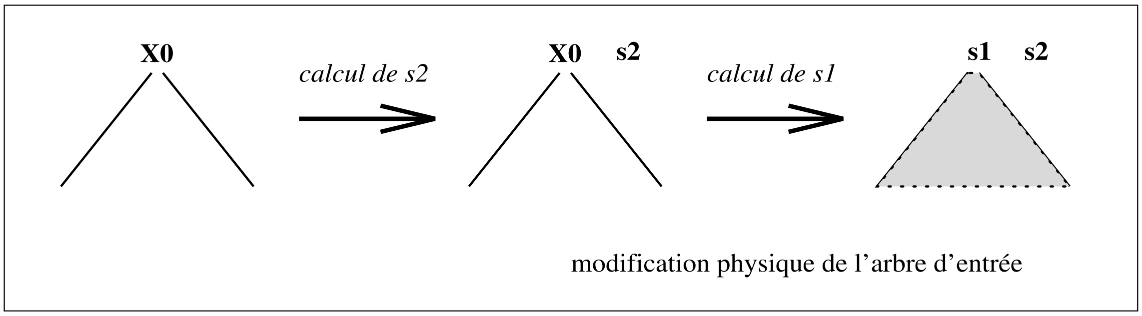


FIG. 5.5 - *Ordre d'évaluation compatible avec la transformation.*

La figure 5.4 présente un ordre d'évaluation (s_1 avant s_2) qui rend un résultat faux. En effet, le calcul de s_1 étant destructif (il modifie l'arbre d'entrée), l'évaluation de s_2 se fait ensuite sur des valeurs qui ne sont plus celles de l'arbre d'entrée.

Par contre, l'ordre d'évaluation (s_2 avant s_1) présenté à la figure 5.5 rend un résultat correct. L'évaluation destructive a été faite en dernier.

Nous voyons ici que l'ordre d'évaluation des attributs est fondamental et qu'il doit être connu par notre algorithme de transformation. Cette information est calculée dans le générateur d'évaluateurs de FNC-2 sous la forme des séquences de visite. L'impératif requis pour qu'un attribut soit évalué destructivement (en modifiant l'arbre d'entrée au nœud courant) est que cette évaluation ait lieu lors de la dernière visite au nœud.

Une visite à un nœud étant un ensemble d'attributs, l'évaluation destructive devra être faite en dernier dans cet ensemble.

La limitation aux attributs synthétisés

Nous pouvons remarquer un cas particulier dans lequel l'évaluation d'un attribut ne se fait pas lors de la dernière visite à un nœud. C'est le cas des attributs hérités. En effet, la particularité d'un attribut de cette classe est d'être calculé en"descendant" le long de l'arbre d'entrée de la grammaire attribuée. En général, ce résultat est utilisé (quelque part) par un attribut synthétisé qui doit "remonter" la structure de l'arbre d'entrée et repasser par le nœud où l'attribut hérité a été évalué. L'évaluation d'un attribut hérité ne se fait donc pas en général lors de la dernière visite à un nœud.

Dans notre algorithme de transformation nous restreindrons donc les évaluations destructives aux attributs synthétisés.

Toutes ces contraintes liées à l'ordre d'évaluation des attributs impliquent un couplage entre notre algorithme de transformation et le générateur d'évaluateurs qui détermine les séquences de visites.

Cela signifie que la transformation que nous allons faire n'est **pas une transformation source à source** d'une grammaire attribuée. Elle prendra sa place au sein du générateur d'évaluateurs et lui fournira des informations pour créer l'évaluateur abstrait.

5.2 L'algorithme

L'algorithme que nous proposons maintenant (cf. figure 5.6) prend en entrée un programme écrit en OLGA qui représente une fonction applicative en grammaire attribuée ; il se place dans le contexte des restrictions présentées précédemment. Les informations qu'il fournit au générateur d'évaluateurs sont symbolisées par un "programme" écrit avec le formalisme d'OLGA augmenté de l'instruction d'affectation `let [...] in [...]`.

Ce résultat représente la version destructive de la fonction applicative d'entrée, dans le contexte d'évaluation choisi[3]. L'utilisation de cette syntaxe OLGA augmentée a pour but de représenter simplement l'effet de la transformation. Ce n'est cependant pas une transformation source à source, puisqu'elle intervient au niveau de l'évaluateur. Nous reviendrons un peu plus en détail sur l'effet de cette transformation et sa situation dans le système FNC-2 dans le chapitre 7.

3. Rappelons que cette transformation utilise les séquences de visites déterminées par le générateur d'évaluateurs ; son résultat est donc lié à un évaluateur.

Pour toute *production* `where cons ->` X_1 ... X_n `use [...]` **faire**

Soit un attribut **S** *tel que*
S soit **synthetisé**
et **S** soit **construit par** un `cons(`val_1`, ... ,`val_n`)`
et **S** soit **évalué** lors de la **dernière visite au noeud courant.**

Si il existe un tel attribut
Alors *(s'il y en a plusieurs, en choisir un ...)*
Remplacer
`cons(`val_1`, ... ,`val_n`) ;`
par
`let`
X_1 `:=` val_1 ;
...
X_n `:=` val_n ;
`in cons ;`
Forcer **S** à être évalué en dernier dans la visite
Fin Si

Fin pour

FIG. 5.6 - *Algorithme de transformation.*

Explications - Commentaires

Cet algorithme effectue dans les cas où cela est possible la même transformation pour chaque règle de production. Il recherche parmi les attributs synthétisés définis dans cette production si il y en a qui sont à la fois construits à partir du même opérateur que celui de la production courante et évalués lors de la dernière visite au nœud courant.

Dans le cas où aucun attribut de cette règle de production ne vérifie ces conditions, l'algorithme ne fait rien (il passe à la règle de production suivante). Au pire, si les attributs évalués n'ont jamais la même structure que les nœuds de l'arbre d'entrée, le résultat de la transformation aboutira à l'identité[4]. Nous pouvons remarquer que les cas les plus intéressants sont ceux où le résultat construit est structurellement proche de l'arbre d'entrée. Les grammaires attribuées concernées par cette optimisation sont celles qui représentent des fonctions mises à jour.

Si il existe des attributs vérifiant les conditions pour la règle de production courante, l'un d'entre eux sera choisi pour avoir une construction destructive. L'algorithme représente

4. Moyennant éventuellement des "free" explicites pour détruire l'arbre d'entrée.

cette action par l'instruction `let [...] in [...]` qui est une information à l'intention du générateur d'évaluateurs. Ce dernier effectuera une affectation physique des éléments en partie droite de la règle de production. Ces éléments sont des variables représentant les descendants du nœud courant. L'instruction `let` sur ces variables représente leur affectation à de nouvelles valeurs. Ainsi, le nœud courant représentera exactement la valeur de l'attribut à construire (sans qu'il y ait eu d'allocation mémoire) et pourra être utilisé en tant que tel.

On prend de plus la précaution (au niveau de l'évaluateur) de forcer cet attribut à être évalué en dernier dans sa visite (qui est déjà la dernière au nœud courant) ; ceci pour être sûr que la valeur de l'arbre d'entrée, qui est modifiée par cette construction, ne sera plus consultée ou utilisée puisqu'elle est désormais invalide.

Cet algorithme utilisé au sein du générateur d'évaluateurs (cf. chapitre 7) permet de produire un évaluateur qui a la sémantique spécifiée par la grammaire attribuée, au même titre qu'un évaluateur classique produit par FNC-2. Cependant, il faut garder à l'esprit qu'il ne peut avoir d'effet qu'à l'intérieur du générateur d'évaluateurs et qu'il n'est pas opérationnel tout seul. Son résultat n'est pas un véritable programme OLGA.

Chapitre 6

La destructibilité de l'argument d'entrée

6.1 La nécessité du critère

Dans le chapitre précédent, nous avons vu comment rendre l'évaluation d'une grammaire attribuée destructive, c'est-à-dire comment faire en sorte que le résultat de la fonction qu'elle représente soit construit en réutilisant la place mémoire allouée pour l'arbre d'entrée.

Nous avons aussi remarqué que l'algorithme ne pouvait être efficace que si la grammaire attribuée transformée représentait une fonction de mise à jour. Il faut en effet que l'attribut construit ait la même structure que l'arbre d'entrée.

Du point de vue du profil des fonctions affectées par cette transformation, il faut que le résultat retourné soit du même type que l'un des arguments :

$$f \ : T_1 \dots T_n \ \rightarrow \ T_i \qquad\qquad \text{où } i \in 1..n$$

Les **fonctions de mise à jour** sont donc les seules qui nous intéressent. Cependant, le cas des fonctions applicatives en général n'est pas incompatible avec l'étude qui suit, puisque dans les cas où $i \notin 1..n$ la transformation n'est rien de plus que l'identité. Nous appellerons *argument d'entrée* l'argument qui est du type du résultat (par analogie avec l'arbre d'entrée d'une grammaire attribuée).

Dans cette section nous nous plaçons dans le cadre d'une application globale, écrite en grammaire attribuée et faisant des appels à des fonctions applicatives. Nous disposons par ailleurs d'une version destructive[1] de chacune de ces fonctions (par l'algorithme de transformation). Nous devons déterminer les conditions dans lesquelles un appel à une fonction peut être remplacé par un appel à sa version destructive.

Après l'appel à une fonction destructive, l'argument d'entrée a été modifié et il ne représente plus la même information qu'avant. Il est donc indispensable que cet objet n'ait plus à être utilisé ultérieurement, que ce soit directement ou indirectement. C'est ce que nous appelons sa **destructibilité**.

1. Eventuellement la même que la version initiale (pour les fonctions autres que de mise à jour).

Pour ce qui est de ses utilisations directes, il suffit de s'assurer qu'il n'est plus nommé après l'appel à la fonction destructive. C'est ce que nous avons déjà vu sous le nom de **durée de vie** (cf. définition 5). Le fait que l'argument d'entrée de la fonction soit **mort** garantit qu'aucune utilisation directe de cet attribut ne sera faite par la suite. Le système FNC-2 permet de calculer cette information qui nous sera donc très utile.

Par contre, le partage de la mémoire (cf. définition 6) qui est induit par les fonctions applicatives pose le problème des utilisations indirectes de l'arbre d'entrée. Il se peut en effet qu'un attribut a soit mort, mais qu'il partage sa structure avec un autre attribut b, bien vivant celui-ci ! Dans ce cas, un appel à une fonction destructive ayant a comme argument d'entrée modifiera (par "effet de bord") la valeur de b. Le résultat de l'application globale sera donc incorrect et c'est ce qu'il nous faut éviter.

L'exemple présenté à la figure 6.1 illustre ce problème. Malgré la fin de la durée de vie de a, cet attribut n'est pas destructible. Le calcul de $a1$ par la fonction destructive Fd peut fausser la valeur de b.

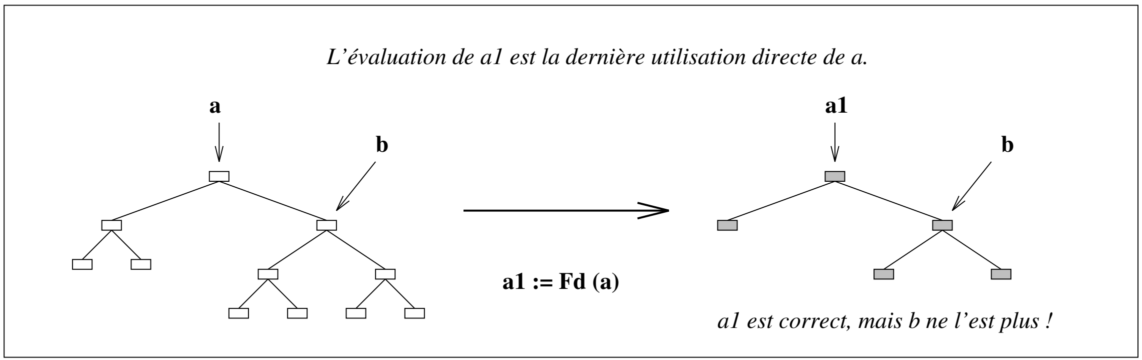


FIG. 6.1 - *Un attribut mort qui n'est pas destructible.*

Nous devons donc calculer sur les attributs de la grammaire attribuée une information de destructibilité qui dépend à la fois de leur durée de vie et de leur partage. Puisque FNC-2 permet déjà de calculer la durée de vie des attributs, nous allons étudier plus spécialement leur partage.

6.2 Utilisation du profil pour l'étude du partage

La question **"l'attribut a est-il partagé ?"** nous amène à distinguer deux manières d'être partagé pour un attribut. Il peut l'être :

- soit parce qu'il a été défini à l'aide d'un attribut dont il utilise une partie de la structure ;
- soit parce qu'il a été utilisé pour définir un autre attribut qui utilise une partie de sa structure.

La notion de partage est la même dans les deux cas : il y a une structure commune aux deux attributs. Cependant il est nécessaire de faire cette distinction pour pouvoir détecter ce partage dans la définition d'une grammaire attribuée.

Dans le premier cas, il s'agit de regarder les attributs qui sont utilisés dans les expressions en partie droite des affectations de a (cf. figure 6.2). C'est un **partage dû à sa définition.**

Dans le second cas, il faut considérer tous les attributs définis par des affectations dont la partie droite contient une occurrence de l'attribut a (cf. figure 6.2). C'est un **partage dû à ses utilisations.**

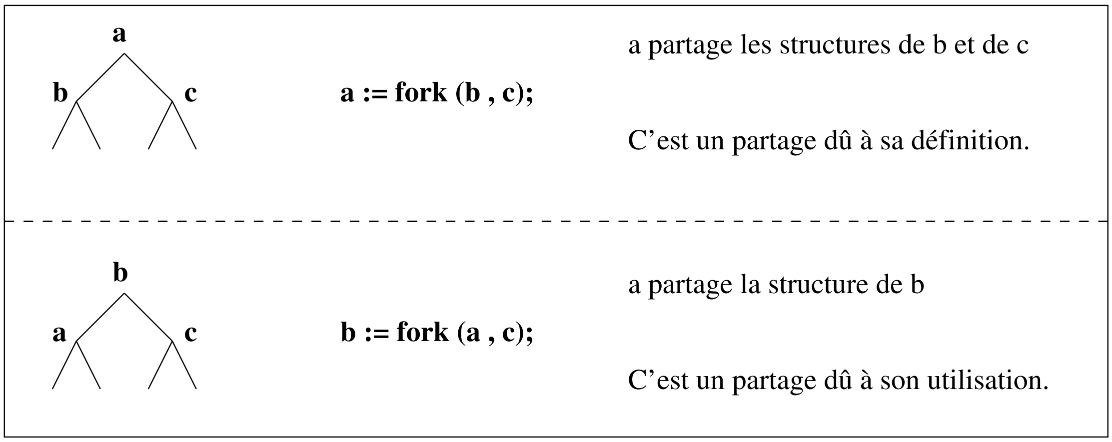


FIG. 6.2 - *Les deux motifs de partage d'un attribut.*

Dans tous les cas, la notion de partage (et de destructibilité) n'est intéressante que si on en a besoin. Or d'après ce que nous avons vu précédemment, cette information est nécessaire sur l'argument d'entrée des fonctions de mise à jour (que l'on veut remplacer par leur version destructive).

Pour effectuer tous les remplacements possibles sur la grammaire attribuée globale, nous aurons donc besoin de connaître l'information de destructibilité sur les arguments d'entrée de toutes les fonctions de mises à jour utilisées[2]. Dans le cas des fonctions de mise à jour, l'argument d'entrée est du même type que le résultat de la fonction.

Définition 8 (Ensemble des types mis à jour) *d'une grammaire attribuée.*

On appelle "types mis à jour" de la grammaire attribuée AG et on note $\mathcal{T}_0(AG)$ l'ensemble des types retournés par les fonctions de mise à jour appelées par AG.

Ceux-ci vérifient donc :

$$T_{i_j} \in \mathcal{T}_0(AG)$$
$$\textbf{si et seulement si}$$
$$\exists\ f \text{ appelée par } AG \text{ et } \exists\ j \in 1..p \text{ tels que}$$
$$f\ : T_1, \ldots, T_n\ \rightarrow\ (T_{i_1}, \ldots, T_{i_p})$$
$$\text{avec } i_j \in 1..n$$

Pour une grammaire attribuée AG donnée, l'ensemble des attributs pour lesquels nous aurons besoin de connaître l'information de destructibilité est inclus dans l'ensemble des

2. Rappelons que pour les autres fonctions la transformation est l'identité. Il ne sera donc pas nécessaire de décider ou non de leur remplacement.

attributs de type appartenant à $\mathcal{T}_0(AG)$. Nous pouvons donc nous limiter à connaître cette information pour les attributs de type mis à jour.

Lors de la détection de la destructibilité de ces attributs, nous nous intéresserons à la façon dont ils sont construits (partage dû à leur définition) et à la façon dont ils sont utilisés (partage dû aux utilisations).

En tant qu'ensemble des types à prendre en compte pour le calcul du partage (et de la destructibilité), on définit l'ensemble $\mathcal{T}(AG)$ de tous les types qui peuvent être influencés par un appel à une fonction de mise à jour.

Cet ensemble contient évidement tous les types qui sont peuvent accepter dans leur structure un type de $\mathcal{T}_0(AG)$ (type mis à jour).

Il contient aussi tous les types qui sont "syntaxiques" dans une fonction de mise à jour, au sens où ils peuvent être analysés dans leur structure (ou construits). Pour déterminer ce dernier ensemble, il suffit de considérer la représentation en grammaire attribuée de la fonction de mise à jour. Elle contient les productions correspondant à l'analyse structurelle de l'argument d'entrée. L'ensemble des types recherché est alors l'ensemble des types des opérateurs (apparaissant en partie gauche des productions `where ... use` du code OLGA de cette grammaire attribuée).

Définition 9 (Ensemble des types syntaxiques) *d'une fonction de mise à jour.*
On appelle "types syntaxiques" d'une fonction de mise à jour l'ensemble des types des non-terminaux apparaissant en partie gauche des productions de la grammaire attribuée représentant cette fonction de mise à jour.

Par exemple, prenons le cas d'une fonction de mise à jour d'une table des symboles implantée par une liste d'arbres binaires de recherche. Chaque nœud de ces arbres contient une information et une clé. La mise à jour va analyser, et éventuellement construire, des listes ou des arbres. Par contre, les types des informations et des clés sont opaques pour la mise à jour. Ils peuvent être manipulés en tant qu'entités, mais ne seront jamais analysés dans leur structure interne et surtout jamais construits. En terme de grammaire attribuée, ce sont des types sémantiques, par opposition aux types liste et arbre qui sont syntaxiques.

Définition 10 (Ensemble des types influencés) *par les mises à jour d'une grammaire attribuée.*
On appelle "types influencés" par les fonctions de mises à jour de la grammaire attribuée AG et on note $\mathcal{T}(AG)$ l'ensemble des types qui sont :

- *soit construits en utilisant dans leur structuredes types de $\mathcal{T}_0(AG)$.*
- *soit syntaxiques dans une fonction de mise à jour de AG.*

Pour reprendre notre exemple d'insertion dans la table des symboles, l'ensemble des types influencés par une grammaire attribuée contient (pour la seule fonction de mise à jour d'insertion) les types arbre, liste et tous les types qui contiennent une table des symboles (en tant que sous-structure).

Dans la section suivante, on décrit un algorithme de calcul de l'information de destructibilité, où le profil de chaque règle sémantique définissant un attribut pourra être restreint en ne considérant que les attributs de type appartenant à $\mathcal{T}(AG)$. En effet, aucun autre attribut ne peut intervenir dans le partage d'un argument d'entrée d'une fonction de mise à jour appelée par la grammaire attribuée considérée : il n'y a donc pas d'intérêt à connaître d'information sur leur destructibilité ou sur leur partage.

Cette réduction de l'ensemble des types des attributs dont on doit calculer la destructibilité constitue une optimisation quant au nombre de calculs à effectuer.

6.3 Un algorithme de décision suffisant

L'algorithme que nous allons proposer ici est dit suffisant parce qu'il permet de garantir qu'une occurrence d'attribut est destructible dans un contexte d'évaluation donné. Il n'est pas optimal, dans ce sens qu'il pourra fournir un résultat de non-destructibilité pour un attribut qui est en fait destructible, mais jamais le contraire. Il est sûr !

Nous travaillons sur une grammaire attribuée AG représentant une application globale. L'algorithme de décision de la destructibilité des occurrences d'attributs (cf. figure 6.3) fonctionne de la même façon que les algorithmes de caractérisation de FNC-2. C'est la propriété d'une occurrence d'attribut qui est calculée d'après les règles sémantiques, production par production, jusqu'à l'obtention de la stabilité de cette propriété sur l'ensemble[3] des occurrences d'attributs de la grammaire attribuée.

A chaque occurrence d'attribut, on associe un booléen $\mathcal{D}$ qui représente sa destructibilité. Puisque la non-destructibilité d'une occurrence d'attribut peut être causée par un partage ou par une utilisation ultérieure, nous initialisons chacune des occurrences $\mathcal{D}$ est à vrai. Ensuite, l'algorithme met à jour cette valeur en fonction des règles sémantiques.

Pour réduire les calculs, nous avons vu précédemment que seules les occurrences d'attributs de type appartenant à $\mathcal{T}(AG)$ pouvaient être détruites physiquement. L'information de destructibilité n'est donc intéressante que sur ces occurrences. Pour plus de lisibilité, l'algorithme que nous présentons ici (figure 6.3) ne tient pas compte de cette restriction. Pour que cette optimisation soit effective il suffit de restreindre l'information de destructibilité et le profil des règles sémantiques aux occurrences d'attributs de $\mathcal{T}(AG)$.

3. Moyennant la restriction à l'ensemble des occurrences d'attributs d'un type appartenant à $\mathcal{T}(AG)$

Le principe

Une occurrence d'attribut peut être modifiée physiquement (est destructible) si elle vérifie ces deux conditions :

- elle est dans son état final (elle n'est plus utilisée ou modifiée par la suite) ;
- elle n'est pas partagée (sa modification n'est pas répercutée sur d'autres valeurs par effet de bord).

Au niveau d'une production, ces conditions sont représentées par la notion de durée de vie d'une occurrence d'attribut. Cette propriété est calculable par FNC-2 et notamment utilisée dans l'optimiseur mémoire du générateur d'évaluateurs.

Nous noterons $\mathcal{L}_{r_{p,a,k}}(X_{f(i)}.a_i)$ le booléen qui est vrai si la règle sémantique $r_{p,a,k}$ ne constitue pas la dernière utilisation de l'occurrence d'attribut $X_{f(i)}.a_i$. Dans ce cas on dit qu'elle est encore en vie. Si c'est la dernière utilisation de cette occurrence d'attribut (elle est morte après cette utilisation), alors le booléen $\mathcal{L}_{r_{p,a,k}}(X_{f(i)}.a_i)$ est à faux.

Moyennant la restriction des profils des règles sémantiques aux types influencés par les mises à jour de la grammaire attribuée, nous pouvons considérer que l'utilisation d'une occurrence d'attribut encore en vie b dans une règle sémantique peut engendrer un partage de l'occurrence définie a. En effet, cette règle sémantique peut utiliser b dans la structure de a et l'utilisation ultérieure de b peut en faire de même avec une autre occurrence d'attribut définie c. Dans ce cas, a et c partageront b.

De la même façon, si une occurrence d'attribut déclarée non destructible est argument d'une règle sémantique, l'occurrence d'attribut définie risque de la partager : elle doit donc également être non destructible.

L'algorithme

L'algorithme est donc basé sur la remarque suivante :
Au niveau d'une production, si une occurrence d'attribut est définie à partir d'occurrences qui sont **toutes destructibles** et **toutes mortes**, alors elle est aussi destructible.

L'extension de cette propriété à toutes les règles sémantiques de toutes les règles de production constitue le principe de l'algorithme. Il réitère la propagation de l'information de destructibilité jusqu'à l'obtention de la stabilité sur l'ensemble des occurrences d'attributs de la grammaire attribuée.

Pour déterminer si une occurrence d'attribut a est destructible, nous étudions la règle sémantique f qui la définit. L'une des deux conditions suivantes est suffisante pour affirmer que a n'est pas destructible :

- f utilise un argument qui est encore en vie ;
- f utilise un argument qui n'est pas destructible.

L'une de ces deux conditions implique donc que $\mathcal{D}(a)$ doit être mis à $false$. Cependant, si l'un des arguments de f est en vie, cela signifie qu'il va être utilisé pour définir une autre occurrence d'attribut. Il risque donc d'engendrer un partage et doit être signalé comme non destructible.

Lorsqu'une occurrence d'attribut devient non destructible, tous les arguments encore en vie de la règle sémantique qui la définit deviennent donc non destructibles.

Pour toute production $p : X_0 \rightarrow X_1 \ldots X_n$ **faire**
  Pour toute occurrence d'attribut $X_k.a$ **faire**
    $\mathcal{D}(X_k.a) := true$
  Fin pour
Fin pour
    fin de l'initialisation

Répéter
  $convergence := true$;
  Pour toute production $p : X_0 \rightarrow X_1 \ldots X_n$ **faire**
    Pour toute règle sémantique $r_{p,a,k} : X_k.a = F_{p,a,k}(X_{f(1)}.a_1, \ldots, X_{f(q)}.a_q)$ **faire**

      Si [$\exists i \in 1..q$ tel que $\neg\mathcal{D}(X_{f(i)}.a_i)$ ou $\mathcal{L}_{r_{p,a,k}}(X_{f(i)}.a_i)$]
      Alors
        Si $\mathcal{D}(X_k.a)$
        Alors
          $\mathcal{D}(X_k.a) := false$
          $convergence := false$;
        Fin Si
        Pour tout $i \in 1..q$ tel que $\mathcal{L}_{r_{p,a,k}}(X_{f(i)}.a_i)$ **faire**
          Si $\mathcal{D}(X_{f(i)}.a_i)$
          Alors
            $\mathcal{D}(X_{f(i)}.a_i) := false$
            $convergence := false$;
          Fin Si
        Fin pour
      Fin Si

    Fin pour
  Fin pour
Jusqu'à $convergence$

FIG. 6.3 - *Algorithme de décision de destructibilité des occurrences d'attributs.*

La figure 6.4 représente un graphe de dépendance global des occurrences d'attributs d'une grammaire attribuée décoré par les informations de destructibilité calculées par notre algorithme de décision.

Pour commenter cet exemple, nous allons supposer que f est évalué avant g. Nous faisons cette hypothèse pour simplifier les explications, mais supposer le contraire aboutirait au même résultat.

La règle sémantique qui définit l'occurrence f utilise i qui n'est pas mort tant que g n'est pas calculé. $\mathcal{D}(f)$ est donc mis à false et $\mathcal{D}(i)$ aussi.

Par la suite, au fil des itérations de l'algorithme, les règles sémantiques qui définissent les occurrences g, c puis a utiliseront des arguments devenus non destructibles. Ces occurrences seront donc signalées comme non destructibles.

Les autres attributs restent destructibles.

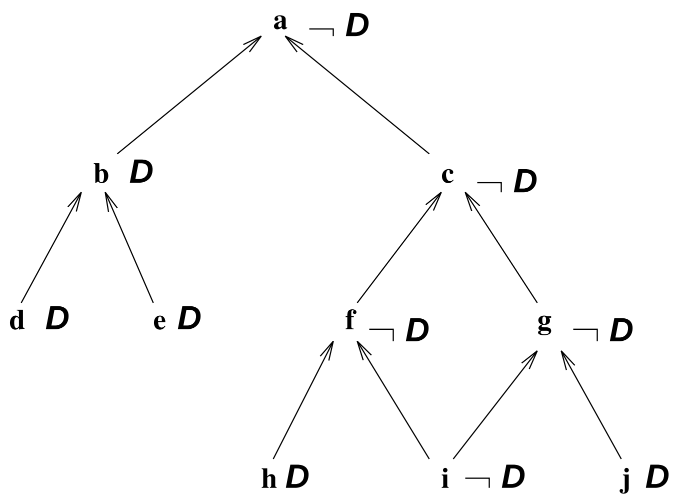


FIG. 6.4 - *Destructibilité des occurrences d'attributs d'un graphe de dépendance.*

Chapitre 7

Mise en place dans Fnc-2

Nous allons dans cette section expliquer comment les algorithmes que nous avons exhibés se mettent en place dans le système Fnc-2. Ces explications se veulent indépendantes du langage de programmation dans lequel ce système est écrit et le moins possible dédiées aux particularités d'implantation. C'est plutôt la situation de ces optimisations au sein du système qui nous intéresse ici et les interactions qu'elles ont avec les autres constituants de Fnc-2.

7.1 L'algorithme de transformation

L'algorithme de transformation destructive de la section 5.2 utilise la syntaxe de la grammaire attribuée fournie par les spécifications Olga et Asx. Il a aussi besoin de connaître l'ordre d'évaluation des attributs choisi, qui est principalement déterminé par les séquences de visites construites à l'intérieur du générateur d'évaluateurs.

Notre algorithme fournit des informations qui doivent être prises en compte pour construire l'évaluateur abstrait. En particulier, à l'intérieur de la dernière visite à un noeud (où un ensemble d'attributs doit être évalué), l'algorithme de transformation peut exiger qu'un attribut précis[1] soit évalué en dernier.

D'autre part, les affectations physiques détruisant la structure de l'arbre d'entrée que nous avions symbolisées par les instructions `let [...] in [...]` doivent être implantées lors du codage des règles sémantiques dans le back end.

L'algorithme de transformation peut donc être mis en place à la fin du générateur d'évaluateurs. Il pourra ainsi utiliser des informations déjà calculées et faire prendre en compte son résultat pour l'élaboration de l'évaluateur abstrait destructif.

La figure 7.1 visualise ces relations dans Fnc-2 entre l'algorithme de transformation et les autres composants :

- Il agit en dernier lieu avant la création de l'évaluateur abstrait ; cela lui permet de forcer l'évaluation d'un attribut à être la dernière d'une visite.

1. Celui qui est construit en détruisant la structure de l'arbre d'entrée.

– Il fournit au back end les renseignements nécessaires pour créer dans le code exécutable les affectations destructives qui sont représentées par les `let [...] in [...]` dans l'algorithme.

Pour une grammaire attribuée représentant une fonction de mise à jour applicative, Fnc-2 fournit donc un évaluateur correspondant à une version destructive de cette fonction.

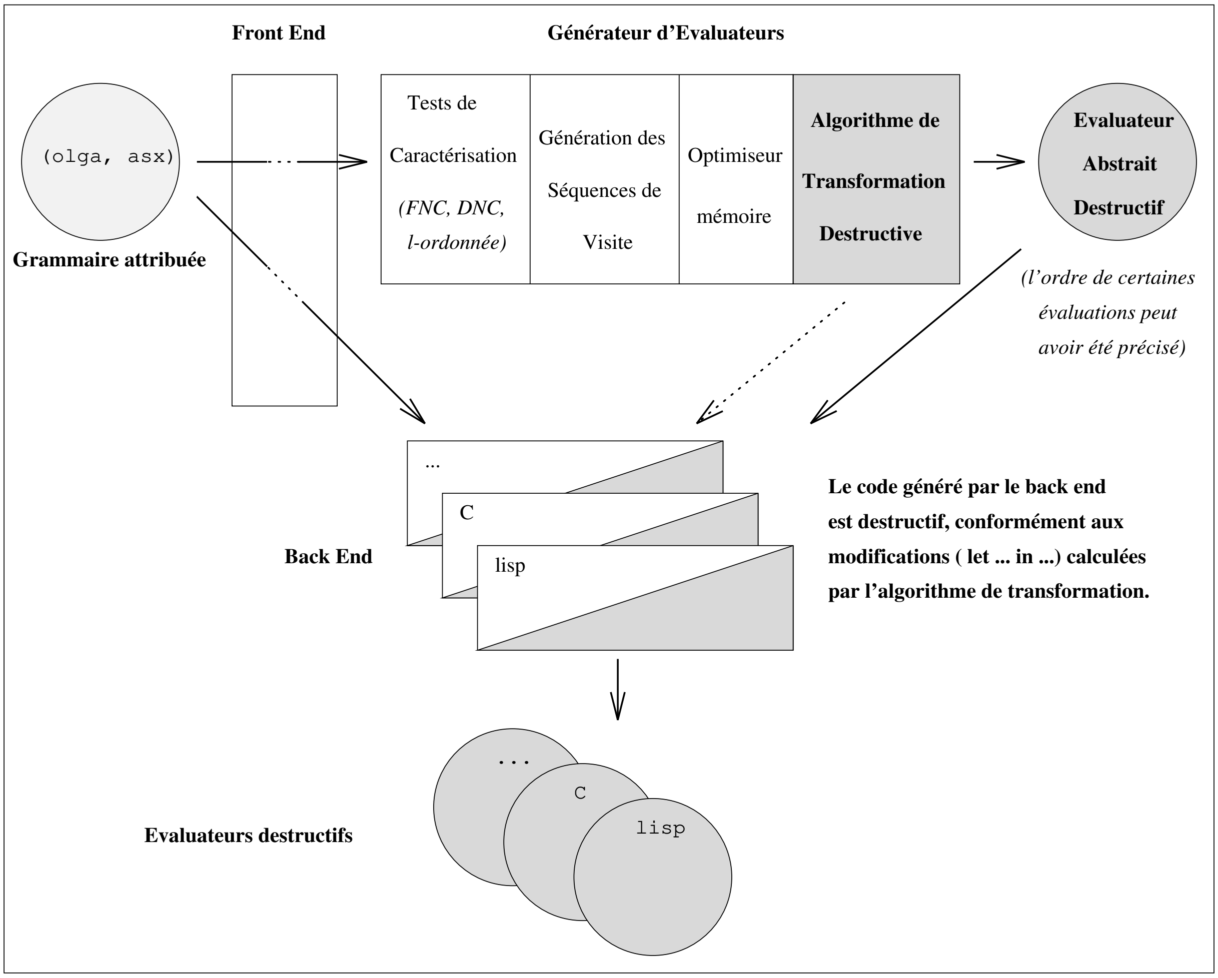


Fig. 7.1 - *Génération de la GA destructive.*

7.2 L'algorithme de décision de destructibilité

Le système Fnc-2 nous permet maintenant de disposer de la version destructive d'une fonction de mise à jour. L'utilisation de cette version destructive dans un contexte particulier d'appel nécessite la destructibilité de son argument d'entrée (cf. section 6.3).

L'algorithme de décision qui permet de déterminer si un attribut est destructible ou non nécessite des connaissances sur la durée de vie des attributs. Ces informations sont déjà calculées par l'optimiseur mémoire du générateur d'évaluateurs de FNC-2 à partir des tests de caractérisation et des séquences de visites.

Pour réutiliser les durées de vie déjà connues, l'algorithme de décision intervient juste après l'optimiseur mémoire (cf. figure 7.2).

Lorsque la grammaire attribuée fait un appel à une fonction de mise à jour, la destructibilité (resp. non-destructibilité) de son argument d'entrée conditionne un appel à la fonction destructive (resp. applicative) dans l'évaluateur abstrait. Celui-ci fait alors référence à l'une ou l'autre, puisqu'elles sont codées distinctement dans le back end.

Le code de l'évaluateur construit par FNC-2 correspondra donc à une version optimisée de la grammaire attribuée fournie en entrée, puisqu'il pourra faire appel à des fonctions destructives (remplaçant des fonctions de mises à jour applicatives).

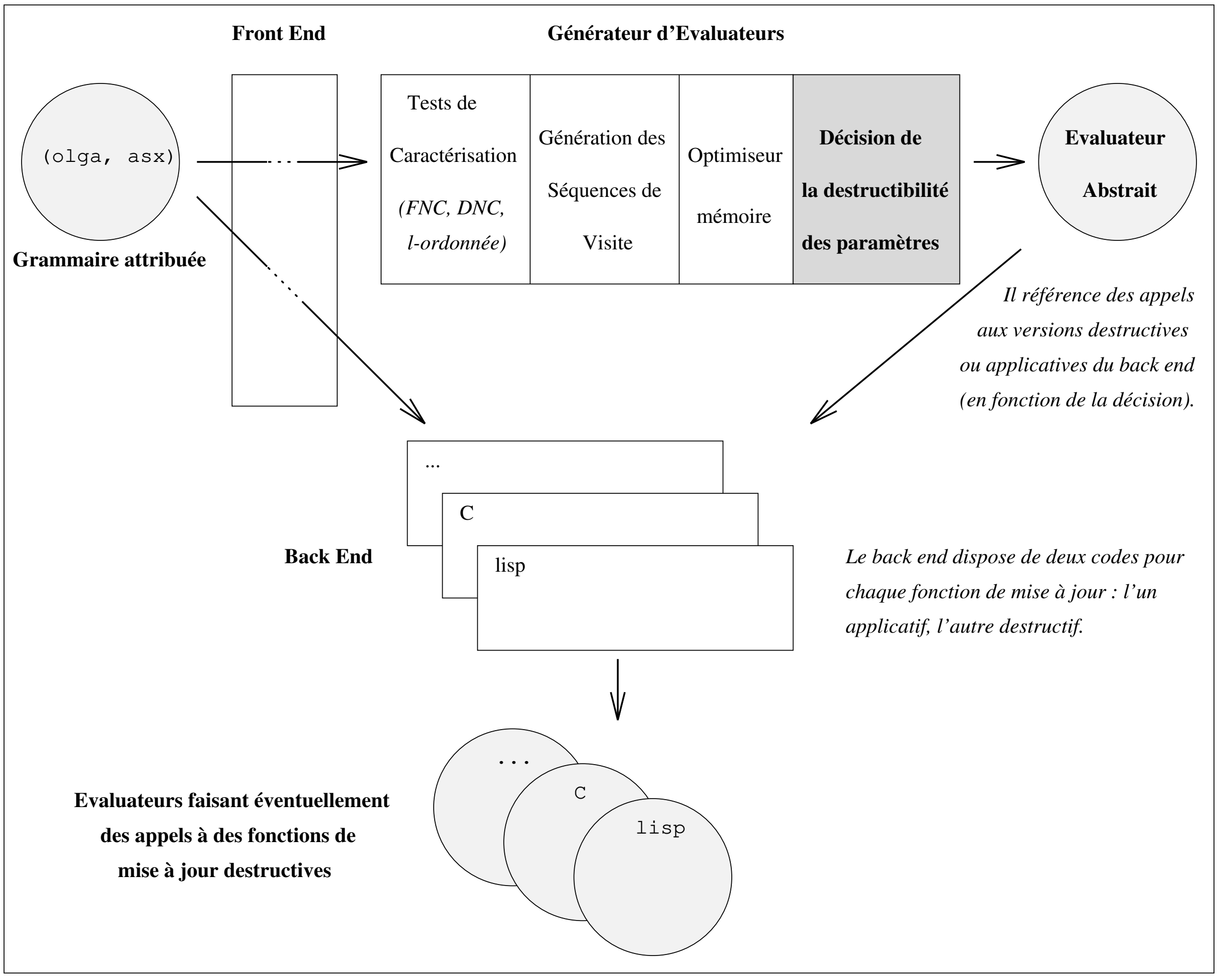


FIG. 7.2 - *Mise en place :* FNC-2 *destructif.*

Chapitre 8

Déforestation et méta-composition

8.1 Le cadre

Le sujet de ce stage étant de transformer des fonctions de mises à jour applicatives en procédures destructives dans un contexte de manipulation de grammaires attribuées, nous nous sommes intéressés au début de cette étude aux liens qui existaient entre le monde fonctionnel et celui des grammaires attribuées. De plus, le but de ce travail étant de réduire l'allocation mémoire faite par le système FNC-2 en utilisant des structures d'arbres, nous avons naturellement été amenés à consulter les travaux qui existaient dans ces domaines, tant du point de vue fonctionnel que du point de vue des grammaires attribuées.

Les travaux de Gilles Roussel [Ro 94], au sein du projet CHLOÉ, sur la *méta-composition* des grammaires attribuées fonctionnelles, traitent de l'optimisation en temps et en place que l'on peut faire en évitant de construire certains arbres (cf. section 8.3). Par ailleurs, dans le cadre des langages fonctionnels, un principe similaire a été étudié par Philip Wadler [Wa 88]. Il s'agit de la *déforestation*, qui consiste à transformer des programmes fonctionnels pour éviter la construction de listes ou d'arbres intermédiaires ([Wa 84], [Wa 85]).

Vu la proximité de ces deux notions, nous avons effectué des observations et des comparaisons sur leurs effets et sur leurs domaines d'action respectifs. Cette section a pour but de présenter les deux méthodes et de les comparer par leur résultats plutôt que par leurs formalismes. Nous ne rentrerons pas dans les détails, mais ferons ici des remarques tendant à rapprocher les grammaires attribuées et la programmation fonctionnelle dans un domaine particulier : la suppression de la construction de structures intermédiaires.

8.2 La déforestation

Les listes intermédiaires et les arbres intermédiaires sont, dans les langages fonctionnels, des liens entre les différentes opérations qui permettent de calculer un résultat. Dans [Wa 88] Philip Wadler donne l'exemple simple suivant :

$$\textit{sum (map square (upto 1 n))} \qquad (1)$$

Ce programme, qui calcule à partir d'un entier n la somme des carrés des entiers de 1 à n utilise, comme **structures intermédiaires** entre les différentes opérations, la liste des entiers de 1 à n, puis la liste des carrés des entiers de 1 à n. Si ces structures semblent ici indispensables au calcul, aucune d'entre elles ne fait réellement partie du résultat. Or elles nécessitent à l'évaluation du résultat l'allocation et la désallocation de la place mémoire qu'elles occupent. De là découle l'idée de Philip Wadler de transformer un programme de sorte que ces structures intermédiaires (listes ou arbres) ne soient pas construites. Son algorithme de déforestation fournit pour cet exemple (1) le programme h suivant :

$$\begin{array}{lll} h\ 0\ 1\ n & & \\ \texttt{where} & & \\ h\ a\ m\ n & = & \texttt{if}\ \ m > n \\ & & \texttt{then}\ \ a \\ & & \texttt{else}\ \ h\ (a + square\ m)\ (m+1)\ n \end{array} \tag{2}$$

Ce programme est plus efficace car toutes les opérations sur les listes ont été éliminées.

Pour présenter cet algorithme nous allons donner quelques définitions, ainsi que la spécification du langage sur lequel Wadler travaille.

Le langage

Nous utilisons ici un langage fonctionnel du premier ordre, avec la grammaire suivante :

$$\begin{array}{lll} t & ::= v & \text{variable} \\ & \mid c\ t_1 \ldots t_k & \text{constructeur} \\ & \mid f\ t_1 \ldots t_k & \text{fonction} \\ & \mid \texttt{case}\ \ t_0\ \ \texttt{of}\ \ p_1 : t_1 \mid \ldots \mid p_n : t_n & \text{terme case} \\ & & \\ p & ::= c\ v_1 \ldots v_k & \text{filtre syntaxique} \\ & & \text{(pattern)} \end{array}$$

Dans une application on appelle $t_1, \ldots, t_n$ des *arguments*. Dans un terme case t_0 est le *sélecteur* et $p_1 : t_1, \ldots, p_n : t_n$ sont les *branches*. Les définitions de fonctions sont de la forme :

$$f\ v_1 \ldots v_k \ = \ t \tag{3}$$

Des exemples de définition de fonction sont présentés dans la figure 8.1.

Habituellement, un terme est dit *linéaire* si il ne contient pas plusieurs fois la même variable. Par exemple, *(append xs (append ys zs))* est linéaire, tandis que *(append xs xs)* ne l'est pas. Cette définition est étendue pour la linéarité des termes case : une variable ne peut pas apparaître à la fois dans une branche et dans le sélecteur, ni plusieurs fois dans une même branche. Par exemple, la définition de *append* est linéaire, même si *ys* apparaît dans chaque branche (cf. figure 8.1).

$$
\begin{array}{lcl}
\textit{list } \alpha & ::= & Nil \mid Cons\ \alpha\ (list\ \alpha) \\
\textit{tree } \alpha & ::= & Leaf\ \alpha \mid Branch\ (tree\ \alpha)\ (tree\ \alpha) \\
 & & \\
append & : & list\ \alpha \rightarrow list\ \alpha \rightarrow list\ \alpha \\
append\ xs\ ys & = & \texttt{case}\ xs\ \texttt{of} \\
 & & \quad \begin{array}{ll} Nil & : ys \\ Cons\ x\ xs & : Cons\ x\ (append\ xs\ ys) \end{array} \\
 & & \\
flip & : & tree\ \alpha \rightarrow tree\ \alpha \\
flip\ zt & = & \texttt{case}\ zt\ \texttt{of} \\
 & & \quad \begin{array}{ll} Leaf\ z & : Leaf\ z \\ Branch\ xt\ yt & : Branch\ (flip\ yt)\ (flip\ xt) \end{array}
\end{array}
$$

FIG. 8.1 - *Exemples de définitions de fonctions.*

La forme treeless

Définition 11 (Forme treeless)

Soit F un ensemble de noms de fonctions. Un terme est dit "treeless" par rapport à F si il est linéaire, si il contient uniquement des fonctions de F et si chaque argument de fonction et chaque sélecteur de terme case qu'il contient est une variable.

Plus formellement, en représentant par tt les termes treeless par rapport à F, on a :

$$
\begin{array}{ll}
tt ::= & v \\
 & \mid c\ tt_1 \ldots tt_k \\
 & \mid f\ v_1 \ldots v_k \\
 & \mid \texttt{case}\ \ v_0\ \ \texttt{of}\ \ p_1 : tt_1 \mid \ \ldots\ \mid p_n : tt_n
\end{array}
$$

avec chaque tt linéaire et chaque f dans F.

Nous dirons qu'une collection de définitions de fonctions F est "treeless" si chaque partie droite dans F (les termes qui définissent les fonctions) est treeless par rapport à F.

Dans la figure 8.1, les définitions de *append* et *flip* sont treeless par rapport à elles-mêmes.

La restriction qui oblige chaque argument de fonction et chaque sélecteur de terme case à être une variable garantit qu'aucun arbre intermédiaire n'est construit. En particulier, cela interdit les termes tels que

$$flip\ (flip\ zt)$$

où *(flip zt)* retourne un arbre intermédiaire. D'autre part, les constructeurs ne sont pas soumis aux mêmes restrictions. Cela autorise les termes tels que

$$Branch\ (flip\ yt)\ (flip\ xt)$$

où les arbres retournés par *(flip yt)* et *(flip xt)* ne sont pas des arbres intermédiaires : ils sont **des parties du résultat**.

La restriction de linéarité garantit que la transformation de certains programmes n'introduit pas de répétitions de calculs. Burstall et Darlington [BD 77] utilisent le terme de "dépliage" *(unfold)* pour décrire le remplacement de la partie gauche d'une équation par la partie droite (comme dans (3), le remplacement d'un nom de fonction par le terme qui la définit).

Un tel dépliage d'une fonction qui a une définition (partie droite) non linéaire risque de dupliquer un terme qui est coûteux à calculer. Par exemple, la fonction non linéaire classique est $square\ x = x * x$. Si t est un terme coûteux à calculer, il est préférable que notre programme contienne $square\ t$ plutôt que son équivalent déplié $t * t$. Par contre, la définition $square\ x = exp(2 * log\ x)$ est linéaire et nous pouvons alors déplier sans risque. La propriété de linéarité nous permet donc de déplier sans perdre d'efficacité.

$$
\begin{array}{lcl}
flatten_0 & : & list\ (list\ \alpha) \rightarrow list\ \alpha \\
flatten_0\ xss & = & \texttt{case}\ xss\ \texttt{of} \\
 & & \quad Nil \qquad\quad\ :\ Nil \\
 & & \quad Cons\ xs\ xss\ :\ append\ xs\ (flatten_0\ xss) \\[1em]
flatten_1 & : & list\ (list\ \alpha) \rightarrow list\ \alpha \\
flatten_1\ xss & = & \texttt{case}\ xss\ \texttt{of} \\
 & & \quad Nil \qquad\quad\ :\ Nil \\
 & & \quad Cons\ xs\ xss\ :\ flatten'_1\ xs\ xss \\[1em]
flatten'_1 & : & list\ \alpha \rightarrow list\ (list\ \alpha) \rightarrow list\ \alpha \\
flatten'_1\ xs\ xss & = & \texttt{case}\ xs\ \texttt{of} \\
 & & \quad Nil \qquad\quad\ :\ Nil \\
 & & \quad Cons\ x\ xs \quad :\ Cons\ x\ (flatten'_1\ xs\ xss)
\end{array}
$$

FIG. 8.2 - *Une définition treeless et une non-treeless pour la même fonction.*

Etre treeless est la propriété d'une définition de fonction et non d'une fonction. La figure 8.2 donne deux définitions d'une même fonction qui "aplatit" une liste de liste en une liste. La définition de $flatten_1$ est treeless, tandis que la définition de $flatten_0$ ne l'est pas. De là le résultat principal proposé par P. Wadler :

Théorème de déforestation

Tout terme linéaire, ne contenant que des occurrences de fonctions ayant des définitions treeless, peut être transformé en un terme treeless équivalent, sans perte d'efficacité[1].

La transformation est effectuée par l'algorithme de déforestation. Si le théorème assure qu'il n'y a pas de perte d'efficacité, il y a en fait un gain lorsque le terme initial contient

1. La sémantique opérationnelle du langage est la réduction de graphe dans l'ordre normal (de gauche à droite et en profondeur d'abord). Un terme est dit plus efficace qu'un autre si, pour chaque instanciation possible des variables libres, le premier requiert moins de pas de réduction que le second.

des arbres intermédiaires. Par exemple, avec les définitions de la figure 8.1, les termes *append* (*append xs ys*) *zs* et *flip* (*flip zt*) satisfont les hypothèses du théorème. Le résultat de l'application de l'algorithme de déforestation sur ces deux fonctions est présenté dans la figure 8.3.

$append\ (append\ xs\ ys)\ zs$

est transformé en

$$
\begin{array}{lcl}
h_0\ xs\ ys\ zs & & \\
\texttt{where} & & \\
h_0\ xs\ ys\ zs & = & \texttt{case}\ \ xs\ \ \texttt{of} \\
 & & \quad Nil \qquad\qquad : \ h_1\ ys\ zs \\
 & & \quad Cons\ x\ xs \quad : \ Cons\ x\ (h_0\ xs\ ys\ zs) \\
h_1\ ys\ zs & = & \texttt{case}\ \ ys\ \ \texttt{of} \\
 & & \quad Nil \qquad\qquad : \ zs \\
 & & \quad Cons\ y\ ys \quad : \ Cons\ y\ (h_1\ ys\ zs)
\end{array}
$$

$flip\ (flip\ zt)$

est transformé en

$$
\begin{array}{lcl}
h_0\ zt & & \\
\texttt{where} & & \\
h_0\ zt & = & \texttt{case}\ \ zt\ \ \texttt{of} \\
 & & \quad Leaf\ z \qquad\quad : \ Leaf\ z \\
 & & \quad Branch\ xt\ yt \quad : \ Branch\ (h_0\ xt)\ (h_0\ yt)
\end{array}
$$

FIG. 8.3 - *Résultats de l'application de l'algorithme de déforestation.*

L'exemple de *append* (*append xs ys*) *zs* est particulièrement intéressant. La représentation d'une exécution présentée dans la figure 8.4 montre que l'utilisation des définitions treeless des fonctions ne suffit pas pour garantir qu'un terme qui utilise ces fonctions le soit également. Plus précisément, dans l'exemple déroulé

$$append\ (append\ [a, b, nil]\ [c, d, nil])\ [e, f, nil]$$

le calcul du premier paramètre du *append* extérieur génère une liste intermédiaire $[a, b, c, d, nil]$ qui est détruite pour reconstruire le résultat final. C'est bien à ce niveau qu'intervient l'algorithme de déforestation de Wadler ; l'exécution (sur le même exemple) de la fonction après l'application de l'algorithme est présentée par la figure 8.5. Dans ce cas, aucune structure n'est construite s'il elle ne fait pas partie intégrante du résultat final.

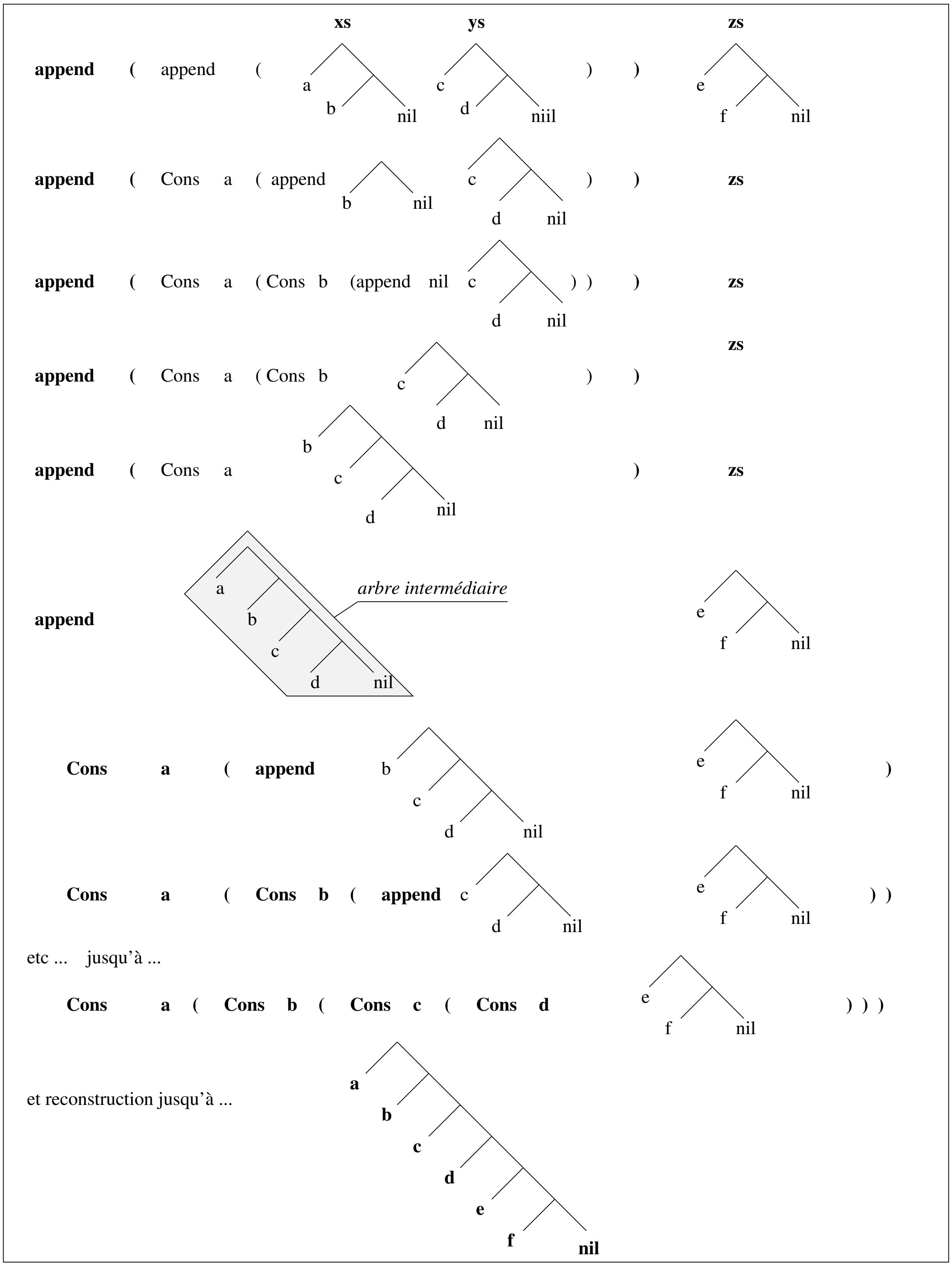


FIG. 8.4 - *Trace d'exécution avec construction d'arbre intermédiaire.*

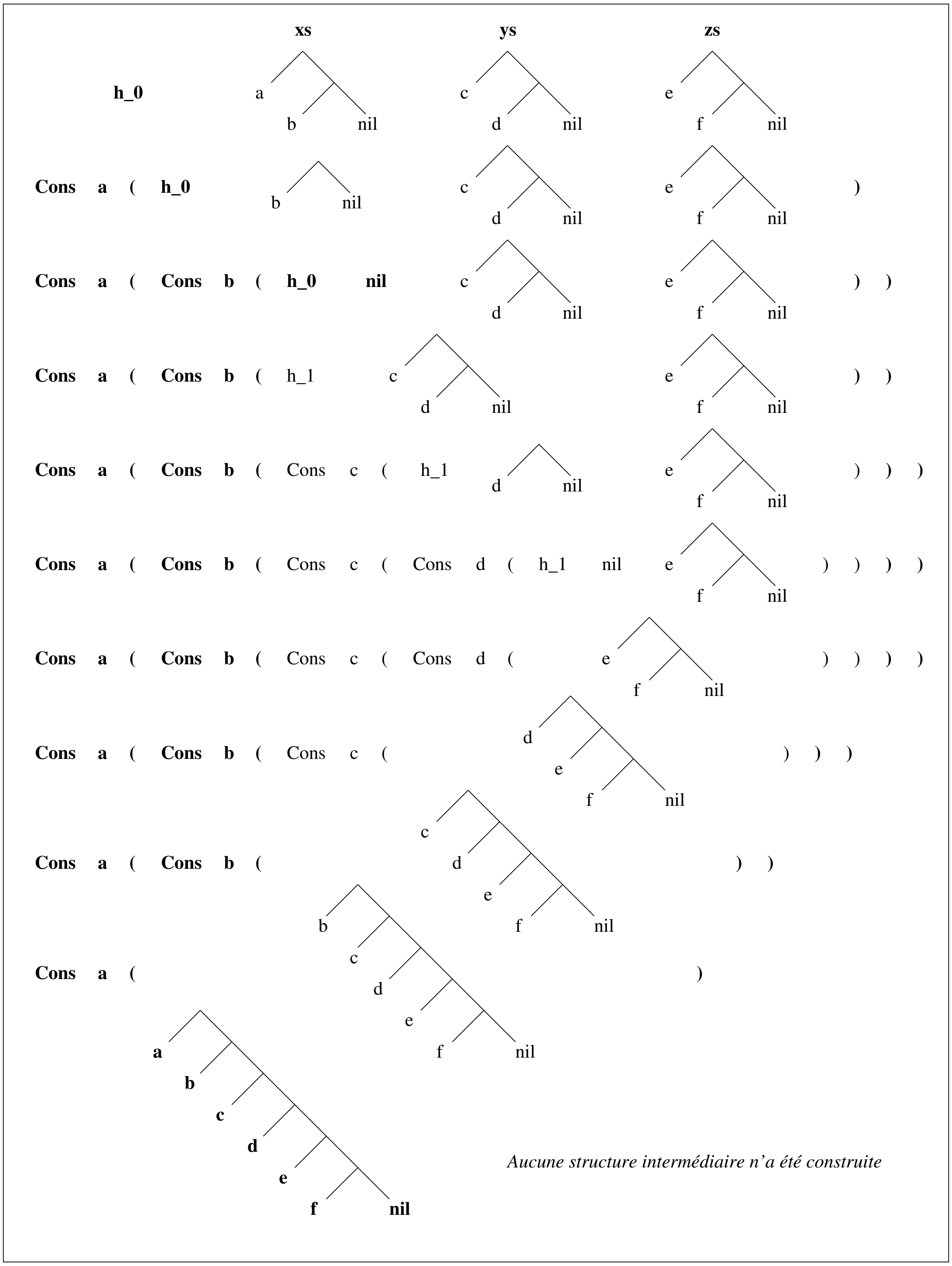


FIG. 8.5 - *Trace d'exécution de la fonction déforestée.*

L'algorithme de déforestation

La caractérisation des définitions treeless et les hypothèses du théorème de déforestation étant purement syntaxiques, il est facile de savoir quand l'algorithme est applicable. Par ailleurs, l'utilisateur n'a pas besoin d'être familier avec les détails de l'algorithme lui-même.

Nous allons donner à la figure 8.6 les sept règles de transformation qui le constituent. Wadler note par $T[\![t]\!]$ le résultat de la conversion d'un terme t dans sa forme treeless. On doit donc avoir :

$$t = T[\![t]\!]$$

c'est à dire que t et $T[\![t]\!]$ doivent calculer la même valeur. Le principe (très général) de l'algorithme est en fait de faire passer les constructeurs vers l'extérieur (vers la gauche) des termes et les fonctions vers l'intérieur. Cela a pour effet de retarder au maximum, lors d'un calcul, la construction d'une structure et d'éviter de la manipuler (par une fonction) après sa construction. Ceci est bien dans l'esprit de la suppression des arbres intermédiaires.

$$
\begin{array}{lll}
(1) & T[\![v]\!] & = v \\[1ex]
(2) & T[\![c\ t_1\ \ldots\ t_k\ ]\!] & = c\ (T[\![t_1]\!])\ \ldots\ (T[\![t_k]\!]) \\[1ex]
(3) & T[\![f\ t_1\ \ldots\ t_k]\!] & = T[\![t[t_1/v_1,\ \ldots,\ t_k/v_k]]\!] \\
 & & \quad \text{où } f \text{ est défini par } f\ v_1\ \ldots\ v_k\ = t \\[1ex]
(4) & T[\![\texttt{case}\ v\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n]\!] & \\
 & & = \texttt{case}\ v\ \texttt{of}\ p'_1 : T[\![t'_1\ ]\!] \mid \ldots \mid p'_n : T[\![t'_n]\!] \\[1ex]
(5) & T[\![\texttt{case}\ c\ t_1 \ldots t_k\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n]\!] & = T[\![t'_i[t_1/v_1,\ \ldots,\ t_k/v_k]]\!] \\
 & & \quad \text{où } p'_i = c\ v_1 \ldots v_k \\[1ex]
(6) & T[\![\texttt{case}\ f\ t_1 \ldots t_k\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n]\!] & \\
 & & = T[\![\texttt{case}\ t[t_1/v_1,\ \ldots,\ t_k/v_k]\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n]\!] \\
 & & \quad \text{où } f \text{ est défini par } f\ v_1\ \ldots\ v_k\ = t \\[1ex]
(7) & T[\![\texttt{case}\ (\texttt{case}\ t_0\ \texttt{of}\ p_1 : t_1 \mid \ldots \mid p_m : t_m)\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n]\!] & \\
 & & = T[\![\texttt{case}\ t_0\ \texttt{of} \\
 & & \qquad p_1\ :\ (\texttt{case}\ t_1\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n) \\
 & & \qquad \ldots \\
 & & \qquad p_m\ :\ (\texttt{case}\ t_m\ \texttt{of}\ p'_1 : t'_1 \mid \ldots \mid p'_n : t'_n)]\!]
\end{array}
$$

FIG. 8.6 - *Règles de transformation de l'algorithme de déforestation.*

La terminaison de l'algorithme se fait grâce à un renomage des expressions qui, lorsqu'elles sont reconnues comme ayant déjà été rencontrées, donnent lieu à des récursions. Les détails des explications de l'algorithme, de sa terminaison et certaines de ses extensions sont données dans [Wa 88].

8.3 La méta-composition

Les grammaires couplées par attributs ont été introduites par Ganzinger et Giegerich [GG 84] pour pouvoir écrire une grammaire attribuée de façon modulaire. Celles-ci sont basées sur une interprétation des grammaires attribuées. L'idée est de ne plus voir une grammaire attribuée comme un processeur qui ajoute des décorations sur un arbre d'entrée, mais comme une véritable fonction qui prend un arbre en entrée et qui produit un autre arbre comme résultat : $t_2 = GA(t_1)$. Pour cette raison, Gilles Roussel appelle ces grammaires couplées par attributs des *grammaires attribuées fonctionnelles* [Ro 94]. De cette façon, il est possible de décomposer une application en une succession de grammaires attribuées fonctionnelles s'appelant en cascade.

La méta-composition [GG 84][Ro 94] est un mécanisme qui permet à partir d'une telle 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.

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.

Nous allons présenter leur principe sans entrer dans les détails de formalisme[2]. Une grammaire attribuée fonctionnelle α fait passer d'un arbre de la grammaire $G_1 = (N_1, T_1, P_1, Z_1)$ à un arbre de la grammaire $G_2 = (N_2, T_2, P_2, Z_2)$. Elle est notée $\alpha : G_1 \to G_2$.

La syntaxe de base de α est la grammaire G_1 et l'attribut synthétisé à la racine est du type de l'axiome Z_2 de G_2.

Le typage des règles sémantiques et l'attribut de la racine du type de Z_2 assurent que le résultat renvoyé par α pourra être considéré comme un arbre de la grammaire G_2.

2. Ce formalisme est présenté dans [Ro 94].

G :

$root : Z \to L$ $cons : L_0 \to Int\ L_1$ $nil : L \to$

inv :

$S(Z) = \{z\}$; $S(L) = \{s\}$; $H(L) = \{h\}$;

$root$: $L.h := nil()$; $Z.z := root(L.s)$;

$cons$: $L_1.h := cons(Int, L_0.h)$; $L_0.s := L_1.s$;

nil : $L.s := L.h$;

FIG. 8.7 - *Grammaire attribuée fonctionnelle spécifiant l'inversion de liste.*

Pour mieux comprendre ces concepts, nous allons nous appuyer sur l'exemple classique de l'inversion de liste [RPJ 95]. La figure 8.7 présente la spécification de la grammaire attribuée fonctionnelle $inv : G \rightarrow G$ qui inverse une liste d'entiers. La grammaire d'entrée $G = (N, T, P, Z)$[3] qui décrit les listes est aussi la grammaire de sortie de inv.

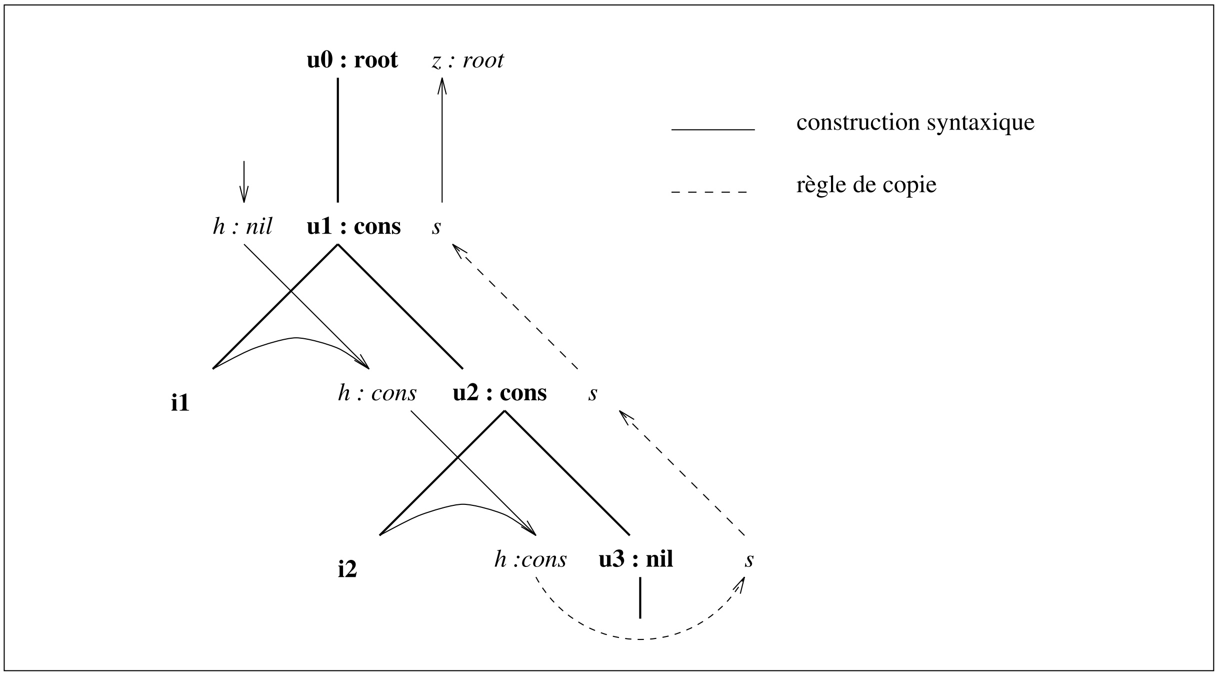


FIG. 8.8 - *L'inversion de liste en action.*

Cette grammaire attribuée fonctionnelle prend en entrée un arbre représentant une liste et construit la liste inverse en utilisant un attribut hérité h. L'attribut synthétisé s ne fait que remonter le résultat à la racine en utilisant des règles de copie.

La figure 8.8 illustre les dépendances et les opérations de inv sur une liste d'entiers à deux éléments. Les constructions syntaxiques d'attributs (dont les opérateurs sont précisés) sont différenciées des règles de copie. Le résultat de la grammaire attribuée fonctionnelle inv est un arbre de la grammaire G.

La méta-composition des grammaires attribuées fonctionnelles

Une application peut être décomposée en une suite de grammaires attribuées fonctionnelles qui se passent des arbres les unes aux autres. Gantzinger et Giegerich ont introduit [GG 84] 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. Cette construction a été étudiée ensuite par Gilles Roussel [Ro 94] sous le nom de *méta-composition*.

3. $N = \{Z, L\}$; $T = \{Int\}$; $P = \{root, cons, nil\}$ décrites dans 8.7.

Prenons deux grammaires attribuées $\alpha : G_1 \rightarrow G_2$ et $\beta : G_2 \rightarrow G_3$. Le but de la méta-composition est de construire une grammaire attribuée fonctionnelle $\gamma : G_1 \rightarrow G_3$ qui ne construit pas l'arbre intermédiaire de G_2.

Elle est notée : $\gamma = \alpha \circ \beta$.

L'idée de base est la suivante : d'une part, les fonctions de constructions de G_2 sont connues dans α et d'autre part, β fournit le schéma de calcul des occurrences d'attributs associées à ces constructions. Il suffit donc de remplacer, avec un renommage, les constructions de G_2 par les schémas de calculs qui leurs sont associés dans β.

L'évaluateur associé à γ effectue alors sur les productions de G_1 où auraient été construits des morceaux d'arbres de G_2, les calculs qui auraient été effectués sur ces morceaux d'arbres par β.

Dans la grammaire attribuée fonctionnelle résultante γ, de nouveaux attributs sont créés. Ils sont nommés en utilisant un attribut de α et un attribut de β.

Les détails de la méta-composition et de sa construction sont détaillés dans [GG 84]. Le formalisme ne nous intéresse pas en lui-même mais nous allons étudier son effet sur notre exemple.

La grammaire attribuée fonctionnelle *inv* (cf. figure 8.7) inverse une liste. Si on l'applique sur son résultat, on retrouve la liste d'entrée (si $inv(l_0)$ donne l_1 et $inv(l_1)$ donne l_2 alors $l_0 = l_2$). Dans ce cas, la liste l_1 est typiquement un arbre intermédiaire.

G :

$root : Z \rightarrow L$	$cons : L_0 \rightarrow Int\ L_1$	$nil : L \rightarrow$

$\gamma = inv \circ inv$:

$S(Z) = \{z.z\}$; $S(L) = \{s.s, h.h\}$; $H(L) = \{s.h, h.s\}$;

$root$: $L.s.h := nil()$;	$cons$: $L_1.s.h := L_0.s.h$;	nil : $L.h.h := L.s.h$;
$L.h.s := L.h.h$;	$L_0.h.h := cons(Int, L_1.h.h)$;	$L.s.s := L.h.s$;
$Z.z.z := root(L.s.s)$;	$L_1.h.s := L_0.h.s$;	
	$L_0.s.s := L_1.s.s$;	

FIG. 8.9 - *Grammaire attribuée fonctionnelle spécifiant* $inv \circ inv$.

La figure 8.9 présente le résultat de la méta-composition de la grammaire attribuée fonctionnelle *inv* avec elle-même. De nouveaux attributs et de nouvelles règles sémantiques ont été créés lors des remplacements des constructions par les schémas de calculs associés.

Par exemple, dans la production $cons : L_0 \rightarrow Int\ L_1$ la règle sémantique $L_1.h := cons(Int, L_0.h)$ induit sur cette production la création des règles sémantiques suivantes :

- $L_0.h.h := cons(Int, L_1.h.h)$ est générée par la projection de la règle sémantique $L_1.h := cons(Int, L_0.h)$,
- $L_1.h.s := L_o.h.s$ est générée par la projection de la règle sémantique $L_0.s := L_1.s$.

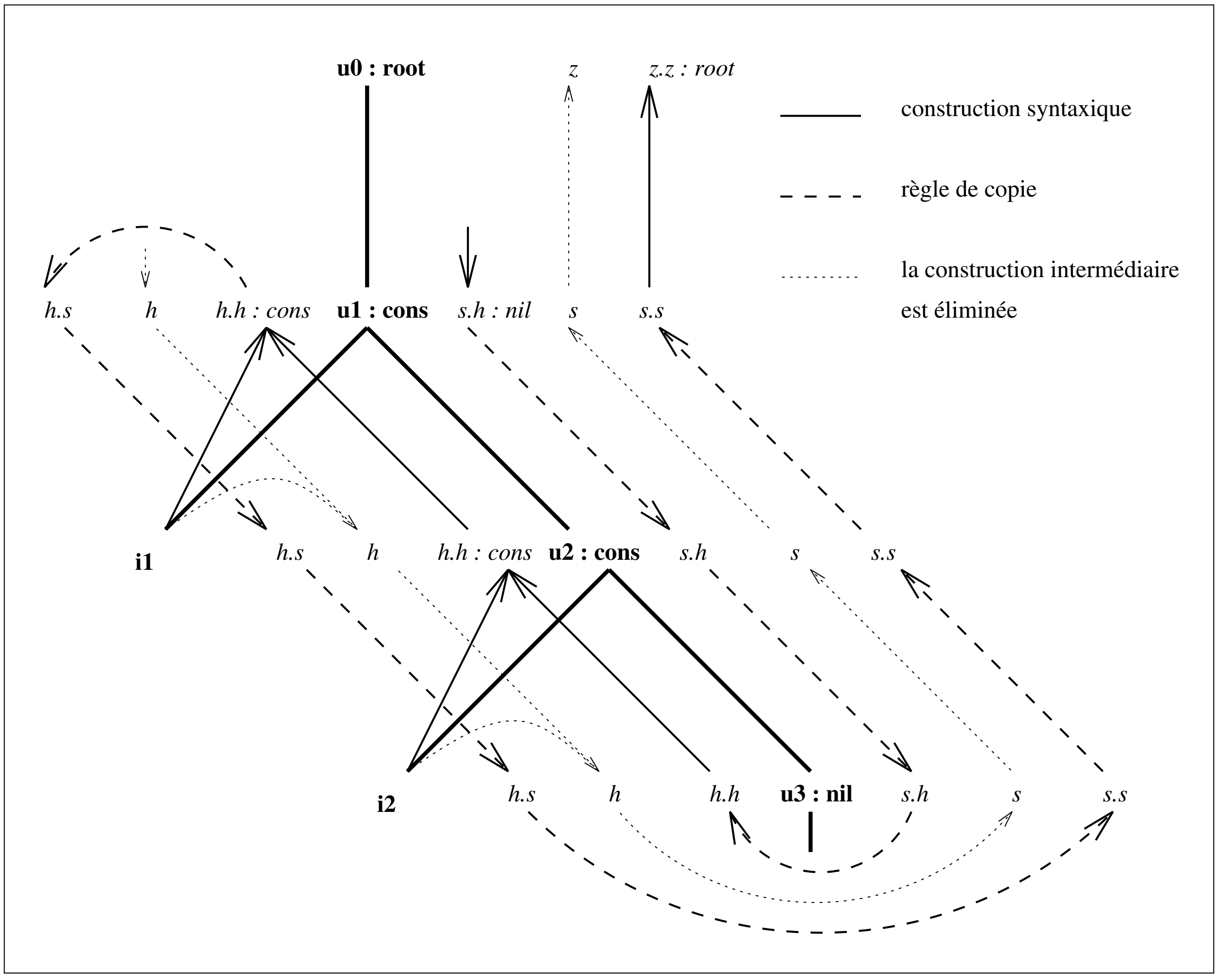


FIG. 8.10 - *La double inversion de liste : inv* $\circ$ *inv.*

Comme nous pouvons l'imaginer la grammaire attribuée fonctionnelle $\gamma = inv \circ inv$ construit une copie de la liste initiale. Pour nous en convaincre, nous avons représenté par la figure 8.10 les dépendances et les opérations de γ sur une liste d'entiers à deux éléments. Les constructions syntaxiques d'attributs y sont différenciées des règles de copies. Nous y avons aussi représenté (en pointillé fin) la construction de l'arbre intermédiaire (l_1) qui a été éliminée.

Cette construction de l'identité peut paraître un peu compliquée. Notons que cette complication tient à l'introduction par la méta-composition de nombreuses règles de copies, qui peuvent être éliminées ultérieurement [Ro 94]. Cette démarche alourdit le calcul statique effectué pour générer l'évaluateur, mais pas l'exécution dynamique, ce qui est le plus important pour nous.

La méta-composition permet donc de **supprimer certaines constructions intermédiaires**.

8.4 Comparaison

Nous connaissons maintenant le principe de la déforestation et celui de la méta-composition. Nous avons pu remarquer qu'ils étaient conçus dans le même esprit de suppression de structures intermédiaires.

Les similitudes

Dans un premier temps, nous allons voir un exemple sur lequel les deux méthodes ont le même effet attendu. Il s'agit du retournement des branches d'un arbre binaire (dont les feuilles sont par exemple des entiers). La figure 8.11 montre le résultat de ce retournement sur un exemple particulier.

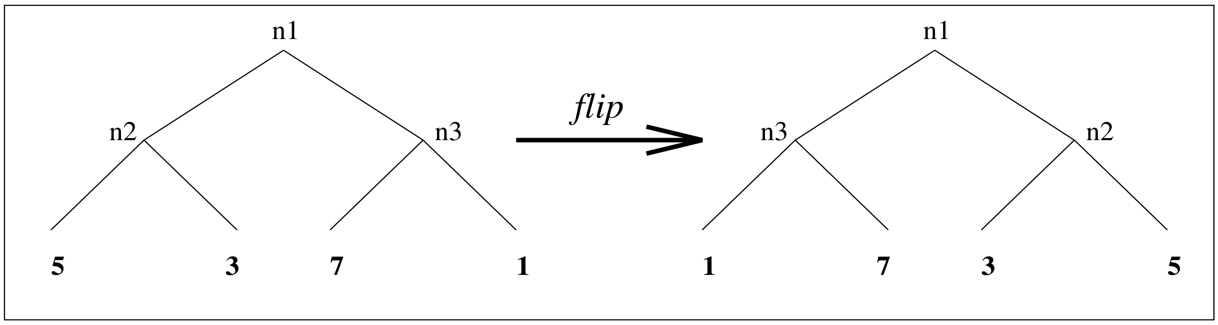


FIG. 8.11 - *Retournement des branches d'un arbre binaire d'entiers.*

Dans le formalisme du langage de Wadler ce retournement de branches que nous appellerons $flip$ est décrit dans la figure 8.1. L'application de l'algorithme de déforestation sur le terme $flip\ (flip\ zt)$ donne le programme présenté à la figure 8.12.

$flip\ (flip\ zt)$

est transformé en

$h_0\ zt$
`where`
$h_0\ zt\ =$ `case` zt `of`
    $Leaf\ z \quad : \quad Leaf\ z$
    $Branch\ xt\ yt \quad : \quad Branch\ (h_0\ xt)\ (h_0\ yt)$

FIG. 8.12 - *Résultats de la déforestation de* $flip\ (flip\ zt)$.

Le double retournement de branches sur un arbre ($flip\ (flip\ zt)$) aboutit naturellement à l'identité. Le programme résultant de la déforestation de ce terme reconstruit simplement l'arbre zt fourni en argument, sans construire l'arbre retourné intermédiaire.

Dans le contexte des grammaires attribuées, ce retournement peut être représentée par la grammaire attribuée fonctionnelle $flip : G \rightarrow G$ où G est la grammaire non-contextuelle qui décrit les arbres binaires d'entiers (cf. figure 8.13).

G :

$root : Z \rightarrow B \qquad branch : B_0 \rightarrow B_1\ B_2 \qquad leaf : B \rightarrow Int$

$flip$:

$S(Z) = \{z\}\,; \qquad S(B) = \{s\}\,;$

$root\ :\ Z.z := root\ (B.s)\,;$
$branch\ :\ B_0.s := branch\ (B_2.s, B_1.s)\,;$
$leaf\ :\ B.s := leaf\ (Int)\,;$

FIG. 8.13 - *Grammaire attribuée fonctionnelle spécifiant le retournement de branches.*

La méta-composition de la grammaire attribuée fonctionnelle $flip$ avec elle-même est présentée dans la figure 8.14. Comme dans le cas de la déforestation, la méta-composition crée une grammaire attribuée fonctionnelle qui reconstruit l'arbre d'entrée sans construire d'arbre intermédiaire.

G :

$root : Z \rightarrow B \qquad branch : B_0 \rightarrow B_1\ B_2 \qquad leaf : B \rightarrow Int$

$flip \circ flip$:

$S(Z) = \{z.z\}\,; \qquad S(B) = \{s.s\}\,;$

$root\ :\ Z.z.z := root\ (B.s.s)\,;$
$branch\ :\ B_0.s.s := branch\ (B_1.s.s, B_2.s.s)\,;$
$leaf\ :\ B.s.s := leaf\ (Int)\,;$

FIG. 8.14 - *Grammaire attribuée fonctionnelle spécifiant $flip \circ flip$.*

Nous sommes en présence d'un exemple qui montre que la déforestation et la méta-composition peuvent avoir exactement le même effet.

Nous savons que la déforestation intervient sur les termes linéaires s'écrivant à partir de fonctions possédant une définition treeless (Théorème de Déforestation, section 8.2).

D'autre part, la méta-composition intervient sur toutes les grammaires attribuées fonctionnelles, moyennant des restrictions explicitées dans [GG 84] et dans [Ro 94]. Cependant, ces grammaires attribuées sont toutes linéaires, et la méta-composition ne requiert aucune restriction similaire à la propriété treeless.

En marge des exemples où tout se passe bien, tant pour la déforestation que pour la méta-composition, nous allons maintenant exhiber deux "contre-exemples".

Un problème pour la méta-composition

Le premier est celui de la concaténation de deux listes ($append\ xs\ ys$). Nous avons étudié cet exemple dans la section 8.2, et nous avons vu qu'un terme tel que $append\ (append\ xs\ ys)\ zs$ pouvait être déforesté en un terme qui ne construit aucune structure intermédiaire.

G :

$root : Z \rightarrow L \qquad cons : L_0 \rightarrow Int\ L_1 \qquad nil : L \rightarrow$

$append$:

$S(Z) = \{z\}\,; \qquad H(Z) = \{l_2\}\,; \qquad S(L) = \{s\}\,; \qquad H(L) = \{h\}\,;$

$root :\ L.h := Z.l_2\,;\ Z.z := root(L.s)\,; \qquad cons :\ L_1.h := L_0.h\,;\ L_0.s := cons(Int, L_1.s)\,; \qquad nil :\ L.s := L.h\,;$

FIG. 8.15 - *Grammaire attribuée fonctionnelle spécifiant la concaténation de deux listes.*

En grammaire attribuée nous pouvons écrire la spécification de $append : G \times G \rightarrow G$ comme le montre la figure 8.15. Le premier argument de $append$ est l'arbre d'entrée de la grammaire attribuée fonctionnelle et le deuxième doit être considéré comme un attribut sémantique de la racine ($Z.l_2$), du type de G. Pour concaténer l_1 à l_2, la grammaire attribuée fonctionnelle $append$ utilise la structure de l'arbre syntaxique représentant l_1 pour diriger le calcul et la valeur sémantique de l_2 pour affecter la valeur de l'attribut hérité à la racine.

La méta-composition de la grammaire attribuée fonctionnelle $append$ avec elle-même semble impossible. En effet, en grammaire attribuée, un impératif est de connaître toute la structure que l'on manipule. Or dans notre exemple ($append \circ append\ l_1\ l_2\ l_3$) il faudrait que la structure de l_2 soit connue pour projeter des règles sémantiques sur ses constructeurs. Par sa nature d'attribut sémantique lors de la première application de $append$, sa structure est ignorée et la méta-composition est donc rendue impossible.

La méta-composition est un mécanisme purement statique et linéaire. Contrairement à la déforestation (qui est un système de réécriture que l'on "déroule") la méta-composition ne permet pas de prendre en compte l'augmentation de la structure d'entrée de $append$, qui passe de l_1 à $l_1.l_2$.

Un problème pour la déforestation

Le deuxième exemple nous est déjà familier : il s'agit de l'inversion de liste, dont nous avons étudié l'effet et la méta-composition avec elle-même dans la section 8.3.

Si nous essayons d'écrire l'inversion de liste avec le formalisme de Wadler, nous obtenons la définition de fonction de la figure 8.16, qui n'est pas treeless. La fonction *inv* prend en entrée la liste à inverser (premier argument) et la liste vide (deuxième argument); elle construit la liste inverse dans le deuxième argument.

$$
\begin{array}{lcl}
inv & : & list\ \alpha \rightarrow list\ \alpha \rightarrow list\ \alpha \\
inv\ xs\ ys & = & \texttt{case}\ xs\ \texttt{of} \\
 & & \quad \begin{array}{lcl} Nil & : & ys \\ Cons\ x\ xs & : & inv\ xs\ (Cons\ x\ ys) \end{array}
\end{array}
$$

FIG. 8.16 - *Définition non treeless de l'inversion de liste.*

La méta-composition nous avait permis de générer une grammaire attribuée qui effectuait la double inversion d'une liste (l'identité) sans construire la liste inversée intermédiaire. Pour en faire autant avec la déforestation, il faudrait disposer d'une définition treeless de l'inversion de liste. A notre connaissance, il n'en existe pas.

Conclusion et travaux futurs

Pour les fonctions restreintes à notre contexte, c'est-à-dire correspondant à des grammaires attribuées qui sont manipulées dans le système FNC-2, la méta-composition est plus intéressante que la déforestation. En effet, ces grammaires attribuées sont toutes linéaires et nous n'avons pas besoin de la notion de treeless.

D'autre part il serait intéressant d'étudier la question suivante : Pour une fonction équivalente à une grammaire attribuée construisant son résultat dans un attribut hérité, est il possible de trouver une définition treeless ?

Le constat que nous pouvons faire est lié au domaine d'application. Sur les grammaires attribuées que nous savons manipuler avec FNC-2, la déforestation semble moins facile à appliquer que la méta-composition. Mais ces grammaires attribuées ne représentent qu'une partie du domaine d'application des langages purement fonctionnels. L'élargissement du langage d'entrée accepté pour les grammaires attribuées permettrait d'affiner la comparaison entre ces deux méthodes.

Chapitre 9

Conclusion

Le cadre de ce stage était le système de traitement de grammaires attribuées FNC-2 et les applications qu'il traite, contenant des appels à des fonctions de mises à jour sur des structures de données abstraites.

Dans ce rapport, nous avons présenté un algorithme qui construit automatiquement la version destructive d'une fonction de mise à jour. Le procédé de construction utilise la représentation en grammaire attribuée de cette fonction et c'est le système FNC-2 qui génère le code destructif correspondant. La version destructive réutilise l'espace mémoire de l'argument d'entrée pour construire le résultat et évite ainsi un grand nombre d'allocations mémoire.

Nous avons également fourni un algorithme capable de décider de la mise en place de ces fonctions de mise à jour destructives à l'intérieur d'une application écrite en grammaire attribuée, en remplacement de leur version applicative. Cet algorithme utilise pour cela les connaissances sur la durée de vie des attributs que FNC-2 est d'ores et déjà en mesure de calculer. En outre, il calcule de façon statique des informations sur la destructibilité et le partage des occurrences d'attributs.

Ces algorithmes peuvent être optimisés et nous avons notamment abordé la notion de restriction du profil des fonctions de mises à jour pour l'étude du partage. Reste que d'autres améliorations sont nécessaires, en particulier pour l'algorithme de décision de destructibilité. En effet ce dernier est suffisant au sens où il est correct ; cependant, il ne permet pas actuellement de traiter certains cas relativement courants, comme nous le montrons dans l'annexe C.

L'implantation de ces algorithmes au sein de FNC-2, dont nous avons brièvement présenté les idées, devrait permettre au système de générer des évaluateurs plus efficaces en temps de traitement et en espace mémoire. Cette future version de FNC-2 disposera alors statiquement de connaissances sur la durée de vie et sur le partage des variables qu'il manipule.

Par ailleurs, nous avons abordé dans ce rapport un problème d'un axe de recherche très peu étudié jusqu'à présent. Il s'agit des liens existant entre la programmation fonctionnelle et le domaine des grammaires attribuées.

Nous avons effectué une présentation et une comparaison de deux méthodes traitant de la suppression des structures intermédiaires : la déforestation et la méta-composition. Les similitudes de comportement de ces deux mécanismes pourraient donner lieu à un rapprochement des deux domaines qui ont chacun, on le sait depuis longtemps, des atouts indéniables, mais qui pourraient s'enrichir mutuellement.

Annexe A

Exemple de transformation destructive

Nous présentons dans cette annexe un exemple illustrant l'effet de notre algorithme de transformation d'une fonction de mise à jour applicative en une fonction destructive. Nous considérons la structure de données représentant la table des symboles dans un langage simple qui n'est pas structuré par blocs (tel que FORTRAN 77 par exemple). La table des symboles pour un tel langage doit disposer des opérations spécifiées ci-dessous :

- la fonction *insert* prend en entrée une clé, une information et une table des symboles et retourne une nouvelle table des symboles dans laquelle a été insérée l'information en fonction de sa clé (elle est mise à jour) ;

- la fonction *already-exist* prend en entrée une clé et une table des symboles et retourne un booléen : vrai si cette clé existe déjà dans la table, faux sinon ;

- la fonction *lookup* prend en entrée une clé et une table des symboles et retourne l'information associée à cette clé. Si la clé ne figure pas dans la table, alors elle retourne une information particulière (disons *key-not-found*[1]).

Nous supposons que les types *symbol-table*, *key-type* et *info-type* sont définis. Supposons également que le type *symbol-table* soit implanté par un arbre binaire de recherche.

Une fonction de mise à jour applicative

La fonction de mise à jour applicative *insert* peut s'écrire sous la forme de la grammaire attribuée *insert-in-tree* qui accepte en entrée un arbre décrit par la syntaxe abstraite *tree*. Les codes ASX et OLGA[2] de cette syntaxe et de cette grammaire attribuée sont présentés par les figures A.1 et A.3.

Nous avons également représenté un exemple d'exécution de cette grammaire attribuée à la figure A.2. Pour être très simple, nous supposons ici que le type *key-type* est le type

1. Cette information peut être gérée par une levée d'exception, par exemple.

2. Nous avons écrit cette grammaire dans l'esprit du langage NEWOLGA qui est en développement actuellement, mais qui n'est pas encore implanté.

chaînes de caractères Olga (*token*) et que *info-type* est le type énuméré { *varid* , *arrayid* } qui renseigne sur la nature de l'identificateur (variable simple ou variable tableau). L'évaluateur de la grammaire attribuée est appelé pour insérer l'information "varid" de clé "G". Notre figure met en évidence le partage occasionné par cette insertion entre la table de symbole fournie en entrée et celle qui est rendue. Les nœuds du chemin de la racine au point d'insertion ont été dupliqués et un nouveau nœud a été alloué pour insérer l'élément "(G,varid)". Cette mise à jour applicative laisse donc l'arbre d'entrée indemne de toute modification physique.

```
{ definition de la syntaxe abstraite TREE }

grammar tree is

   root is TREE;

   TREE   =  node null;

   node   -> key-type info-type TREE TREE;
   null   -> ;

end grammar;
```

Fig. A.1 - *Syntaxe abstraite de l'arbre binaire de recherche :* `tree.asx`

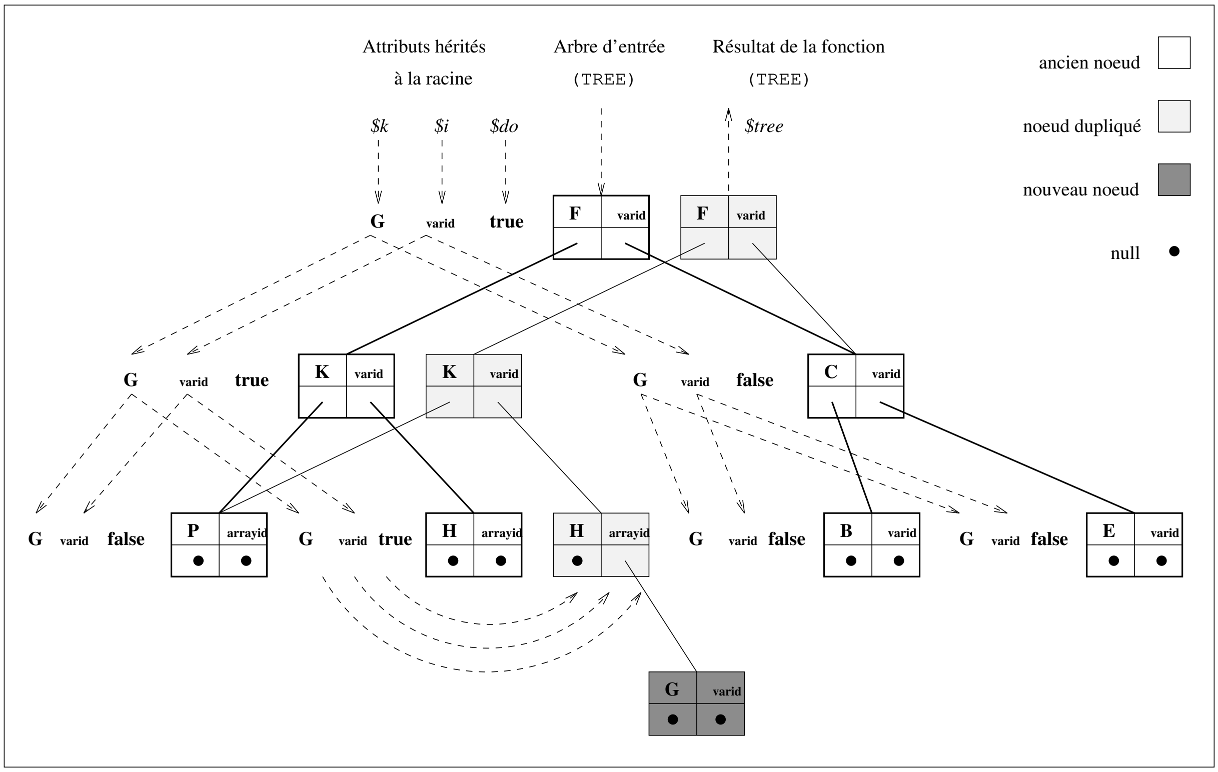


Fig. A.2 - *L'effet de insert-in-tree sur un exemple particulier.*

```
{ grammaire attribuee INSERT_IN_TREE }

attribute grammar insert-in-tree ($k(root)  : key-type,
                                  $i(root)  : info-type,
                                  $do(root) : bool := true,
                                  TREE                    ) : (TREE) is

attribute

   inherited   $k(TREE)    : key-type;
   inherited   $i(TREE)    : info-type;
   inherited   $do(TREE)   : bool;
   synthesized $tree(TREE) : TREE;


{ On suppose que leaf() est une fonction qui alloue une feuille vide     }
{ (non partagee) et qui la retourne.                                     }

{ fake-tree est une fonction qui retourne une valeur bidon (non partagee)}

where node -> KEY INFO LEFT RIGHT use
   $k(LEFT) := $k(node);
   $k(RIGHT) := $k(node);
   $i(LEFT) := $i(node);
   $i(RIGHT) := $i(node);
   $do(LEFT) := (KEY < $k(node));        { marque la branche d'insertion }
   $do(RIGHT) := (KEY > $k(node));
   $tree := if $do(node) then       { on est dans la branche d'insertion }

               if (KEY = $k) then                 { insertion a ce noeud }
                  node(KEY,$i(node),LEFT,RIGHT)
               else
                    if (KEY < $k(node)) then
                                       { insertion dans la branche gauche }
                       node(KEY,INFO,$tree(LEFT),RIGHT)
                    else               { insertion dans la branche droite }
                       node(KEY,INFO,LEFT,$tree(RIGHT))
                    end if
               end if
            else
               fake-tree()                           { valeur inutilisee }
            end if;
end where;

where null -> use
   $tree := if $do (null) then                 { ajout de cette feuille }
               node($k(node),$i(node),leaf(),leaf())
            else
               fake-tree()                           { valeur inutilisee }
            end if;
end where;

end grammar {insert-in-tree};
```

Fig. A.3 - *Grammaire attribuée représentant la fonction insert :* `insert-in-tree.olga`

La version destructive de cette fonction de mise à jour

L'algorithme de transformation que nous avons présenté au chapitre 5.2 prend en entrée la grammaire attribuée `insert-in-tree.olga` de la figure A.3 et agit au niveau du générateur d'évaluateurs d'FNC-2. La figure A.5 montre les productions et les règles sémantiques produites par notre algorithme. Elles représentent les informations qui sont fournies au générateur d'évaluateurs et au back end.

Afin de visualiser la différence avec la version applicative, nous avons repris à la figure A.4 l'exemple de l'insertion de l'information "varid" de clé "G" exécuté avec la version destructive.

Nous pouvons voir que la structure du résultat retourné est physiquement celle de l'arbre d'entrée, dont certaines valeurs ont changé. Les seules allocations qui ont été faites correspondent au nouveau nœud inséré, dont la clé n'existait pas dans l'arbre d'entrée. Remarquons également que cette fonction n'introduit plus de partage. Nous avons donc ici un évaluateur destructif qui fait de l'*update in place* (ou mise à jour en place).

L'optimisation que procure cette version par rapport à la version applicative est à la fois temporelle et spatiale. L'algorithme qui la produit utilise bien sûr du temps et de la place mémoire, mais de façon statique, au moment de la compilation et une fois pour toutes. En outre, il évite les allocations mémoires qui dupliquaient une partie de la structure.

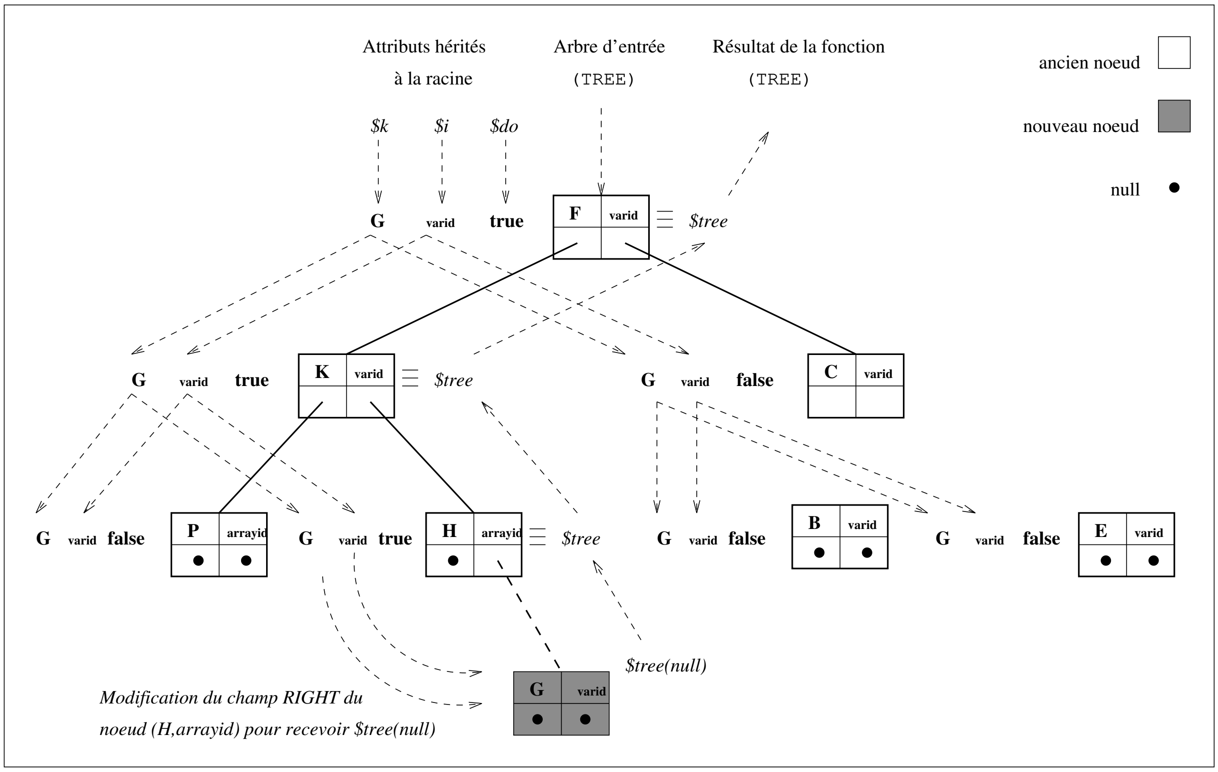


FIG. A.4 - *L'effet de la version destructive sur notre exemple.*

```
where node -> KEY INFO LEFT RIGHT use
   $k(LEFT) := $k(node);
   $k(RIGHT) := $k(node);
   $i(LEFT) := $i(node);
   $i(RIGHT) := $i(node);
   $do(LEFT) := (KEY < $k(node));
   $do(RIGHT) := (KEY > $k(node));

{ L'attribut $tree verifie les hypotheses de l'algorithme de }
{ transformation. Il est le seul de la derniere visite au    }
{ noeud node, donc il est le dernier de cette visite.        }
{ Partout ou il y a mapping syntaxique on va reutiliser la   }
{ structure de l'arbre d'entree pour construire l'attribut   }
{ synthetise.                                                }
{ La valeur de $tree prend alors celle du noeud modifie node }

   $tree := if $do(node) then
               if (KEY = $k) then
                  let
                     INFO := $i(node)
                  in node;
               else if (KEY < $k(node)) then
                       let
                          LEFT := $tree(LEFT)
                       in node;
                    else
                       let
                          RIGHT := $tree(RIGHT)
                       in node;
                    end if
               end if
            else
               fake-tree()                   { valeur inutilisee }
            end if;
end where;

where null -> use
{ Ici, il n'y a pas mapping syntaxique, donc pas de reutilisation }

   $tree := if $do (null) then
               node($k(node),$i(node),leaf(),leaf())
            else
               fake-tree()                   { valeur inutilisee }
            end if;
end where;
```

FIG. A.5 - *Règles sémantiques "destructives" de* `insert-in-tree.olga`

Annexe B

Exemple de mise en place

Nous voulons maintenant présenter une application écrite en grammaire attribuée qui utilise la fonction de mise à jour applicative *insert*. Cette application est la phase de vérification sémantique d'un compilateur. La taille et la complexité d'un compilateur d'un langage tel que FORTRAN 77 ne nous permettent pas de présenter ici l'application dans son entier, mais nous allons essayer d'en présenter la partie qui nous intéresse plus particulièrement.

Nous allons considérer un langage de programmation “d'école” que nous appellerons RoutineSimple. Nous nous sommes inspirés de l'exemple du langage Simproc du manuel de référence de Fnc-2 [JP 89], mais nous voulions un langage non structuré par bloc (ce qui n'est pas le cas de Simproc)[1].

Les grands traits de RoutineSimple sont les suivants (le lecteur pourra se reporter à [JP 89] pour les autres détails) :

- il n'est pas structuré par blocs ;
- il n'y a pas de procédure ni de fonction ;
- il doit y avoir unicité de la déclaration des identificateurs ;
- chaque identificateur doit respecter dans ses utilisations les contraintes de sa déclaration (variable simple, variable tableau ...).

D'autre part, il n'y a pas de vérification de type : toutes les variables et les expressions sont du type entier.

La phase d'analyse syntaxique d'un programme source écrit en RoutineSimple construit un arbre que l'on peut décrire par la syntaxe abstraite (`routine-simple.asx`) présentée dans la figure B.1.

1. Nous nous expliquerons sur ce point dans l'annexe C.

```
{ syntaxe abstraite de notre langage: programme routine-simple.asx }

grammar routine-simple is

   root is ROUTINE;

   { routine }

   ROUTINE     =  routine;
   routine     -> DECLS STMTS;

   { declarations }

   DECLS       =  decls end-decl;
   decls       -> DECL DECLS;
   end-decl    ->;
   DECL        =  int-decl array-decl;
   int-decl    -> ID  { identificateur de variable }
   array-decl  -> ID  { identificateur de tableau }
                  NUMBER { la dimension };

   { statements }

   STMTS       =  stmts end-stmt;
   stmts       -> STMT STMTS;
   end-stmt    -> STMT;
   STMT        =  assign ifthenelse;
   assign      -> VAR EXPR;
   ifthenelse  -> EXPR EXPR  { la seule comparaison est l'egalite }
                  STMT { partie true } STMT { partie false };

   { expressions }

   EXPR        =  bin-expr constant VAR { inclusion de phylum };
   bin-expr    -> OP EXPR EXPR;
   constant    -> NUMBER;
   OP          =  plus minus mul div;
   plus        -> ;
   minus       -> ;
   mul         -> ;
   div         -> ;

   { variables }

   VAR         =  simple-var indexed-var;
   simple-var  -> ID  { identificateur de variable }
   indexed-var -> ID  { identificateur de tableau }  EXPR  { index }

   { terminaux }

   NUMBER      =  number;
   number      -> ;
   ID          =  id;
   id          -> ;

end grammar; {routine-simple}
```

FIG. B.1 - *Syntaxe abstraite de* ROUTINESIMPLE *:* `routine-simple.asx`

La phase de vérification sémantique

Nous allons maintenant décrire la grammaire attribuée qui prend en entrée un arbre de la syntaxe abstraite `routine-simple.asx`, représentant un programme écrit en Routine-Simple et qui effectue la phase de vérification sémantique de ce programme. Elle doit donc construire la table des symboles en vérifiant qu'il n'y a pas de double déclaration ; pour cela, elle doit parcourir toutes les déclarations du programme. Une fois cette table construite, elle doit s'en servir pour vérifier que les utilisations des identificateurs qui sont faites dans la partie "statements" du programme sont conformes à leur déclaration.

L'avantage de cet exemple est qu'il permet de présenter une application écrite en Olga qui est cohérente. Cependant, pour ne pas perdre le lecteur dans des considérations qui ne sont pas les nôtres ici, nous avons "idéalisé" cette application, en considérant que tout se passe bien et qu'on ne retourne aucun message d'erreur. Encore une fois, l'exemple Simproc de [JP 89] fournit des détails plus proches de la réalité d'un compilateur.

La grammaire attribuée `check-rout-simp` retourne donc un booléen qui est vrai si les définitions et les utilisations des identificateurs sont sémantiquement conformes à la spécification du langage.

Les règles sémantiques utilisent sur le type *symbol-table* les fonctions applicatives *lookup*, *already-exist* et *insert*. Notons que cette dernière est la fonction de mise à jour applicative dont nous avons montré la transformation destructive dans l'annexe A. Nous supposons que la valeur *empty-table* est définie comme représentant un arbre binaire de recherche vide et qu'elle n'est pas partagée.

Nous avons repéré par une flèche en commentaire chacun des appels à une fonction qui manipule un attribut de type *symbol-table*. Les autres manipulations sur ce type sont des règles de copie.

```
{ grammaire attribuee qui effectue la verification semantique   }
{ d'un programme RoutineSimple represente par un arbre abstrait }
{ de routine-simple.asx                                         }

attribute grammar check-rout-simp (routine-simple) : bool is

attribute

   inherited   $symtab (STMTS, STMT, EXPR,  { represente la table des symboles completement construite }
               OP, VAR, ID, NUMBER)       : symbol-table;  { qui est heritee dans la partie statements }

   inherited   $h-temp-st (DECLS, DECL)   : symbol-table;      { utilises pour construire la table des }
   synthesized $s-temp-st (DECLS, DECL)   : symbol-table;      { symboles dans la partie declaration   }

   synthesized $correct (ROUTINE)         : bool;                 { resultat dela grammaire attribuee }

   synthesized $correct1 (DECLS, DECL)    : bool;   { represente la correction de la partie declaration}
   synthesized $correct2 (STMTS, STMT,
               EXPR, OP, VAR, ID, NUMBER) : bool;   { represente la correction de la partie declaration}

   synthesized $id (ID)                   : token;
   synthesized $value (NUMBER)            : int;
```

```
{ regles semantiques pour chaque production }

{ routine }

where routine -> DECLS STMTS use
   $h-temp-st(DECLS) := empty-table;
   $symtab(STMTS) := s-temp-st(DECLS);
   $correct := $correct1(DECLS) & $correct2(STMTS);
end where;

{ declarations }

where decls -> DECL DECLS use
   $correct1 := $correct1(DECL) & $correct1(DECLS);
   $h-temp-st(DECL) := $h-temp-st(decls);
   $h-temp-st(DECLS) := $s-temp-st(DECL);
   $s-temp-st(decls) := $s-temp-st(DECLS);
end where;

where end-decl -> use
   $correct1 := true;
   $s-temp-st := $h-temp-st;
end where;

where int-decl -> ID  use
   declare
      correct := if $id(ID) = (error-token) or
                   already-exist($id(ID), $h-temp-st)          {  <----------}
                            then
                    false
                 else
                    true
                 end if;
   use
      $correct1 := correct;
      $s-temp-st := if correct then
                       insert($id(ID), varid, $h-temp-st)      {  <----------}
                    else
                       $h-temp-st
                    end if;
end where;

where array-decl -> ID NUMBER use
   declare
      correct := if ($id(ID) = error-token
                  or already-exist($id(ID), $h-temp-st)        {  <----------}
                  or $value(NUMBER) = 0) then
                    false
                 else
                    true
                 end if;
   use
      $correct1 := correct;
      $s-temp-st := if correct then
                       insert($id(ID), arrayid, $h-temp-st)    {  <----------}
                    else
                       $h-temp-st
                    end if;

end where;

{ statements }

where stmts -> STMT STMTS use
   $symtab(STMT) := $symtab(stmts);
   $symtab(STMTS) := $symtab(stmts);
   $correct2 := $correct2(STMT) & $correct2(STMTS);
```

```
end where;

where end-stmt -> STMT use
   $symtab(STMT) := $symtab(stmts);
   $correct2 := $correct2(STMT);
end where;

where assign -> VAR EXPR use
   $symtab(VAR) := $symtab(assign);
   $symtab(EXPR) := $symtab(assign);
   $correct2 := $correct2(VAR) & $correct2(EXPR);
end where;

where ifthenelse  -> EXPR1 EXPR2 STMT-T STMT-F use
   $symtab(EXPR1) := $symtab(ifthenelse);
   $symtab(EXPR2) := $symtab(ifthenelse);
   $symtab(STMT-T) := $symtab(ifthenelse);
   $symtab(STMT-F) := $symtab(ifthenelse);
   $correct2 := $correct2(EXPR1) & $correct2(EXPR2) &
                $correct2(STMT-T) & $correct2(STMT-F);
end where;

{ expressions }

where bin-expr -> OP EXPR1 EXPR2 use
   $symtab(EXPR1) := $symtab(bin-expr);
   $symtab(EXPR2) := $symtab(bin-expr);
   $correct2 := $correct2(EXPR1) & $correct2(EXPR2);
end where;

where constant -> NUMBER use

end where;

where EXPR -> VAR use
   $symtab(VAR) := $symtab(EXPR);
   $correct2 := $correct2(VAR);
end where;

where plus -> use

end where;

where minus -> use

end where;

where mul -> use

end where;

where div -> use

end where;

{ variables }

where simple-var -> ID use
   $correct2 := if ($id(ID) = error-token) or
                 (lookup($id(ID), $symtab) = key-not-found)    {  <----------}
                              then
                   false
                elsif lookup($id(ID), $symtab) = arrayid       {  <----------}
                              then
                   false
                else
                   true
```

```
              end if;

end where;

where indexed-var -> ID EXPR use
   $symtab(EXPR) := $symtab;
   $correct2 := (if ($id(ID) = error-token) or
                   (lookup($id(ID), $symtab) = key-not-found)   {  <----------}
                                then
                   false
                elsif lookup($id(ID), $symtab) = varid        {  <----------}
                                then
                   false
                else
                   true
                end if;) &
                $correct2(EXPR);
end where;

{ terminaux }

where number -> use
   $value(NUMBER) := {valeur de l'entier}
end where;

where id -> use
   $id(ID) := {valeur de l'identificateur (type token)}
end where;


end grammar; {check-rout-simp}
```

Le remplacement de *insert* par sa version destructive

Nous avons dans cette grammaire attribuée deux règles sémantiques qui contiennent des appels à la fonction de mise à jour applicative *insert*. Nous voulons donc maintenant illustrer l'effet de notre algorithme de décision de la section 6.3, qui permet de mettre en place la version destructive de *insert* si son argument d'entrée est destructible.

La fonction de mise à jour *insert* est appelée lors de la déclaration d'une variable ou d'un tableau

`int-decl ->ID` ou `array-decl ->ID NUMBER`.

Elle prend à chaque fois l'attribut `$h-temp-st` en argument pour construire l'attribut `$s-temp-st`.

Nous n'allons pas donner la trace d'exécution du système FNC-2 qui construit l'évaluateur associé à cette grammaire attribuée, mais nous pouvons constater que sur une instance d'un nœud `DECL` de l'arbre d'entrée, l'appel à *insert* constitue la dernière utilisation de l'occurrence d'attribut `DECL.$h-temp-st`. De plus, aucune des instances d'attributs `$h-temp-st` ou `$s-temp-st` n'est partagée. Nous avons présenté à la figure B.2 une partie du graphe de dépendances d'un exemple de deux déclarations que nous avons décoré par l'information de destructibilité.

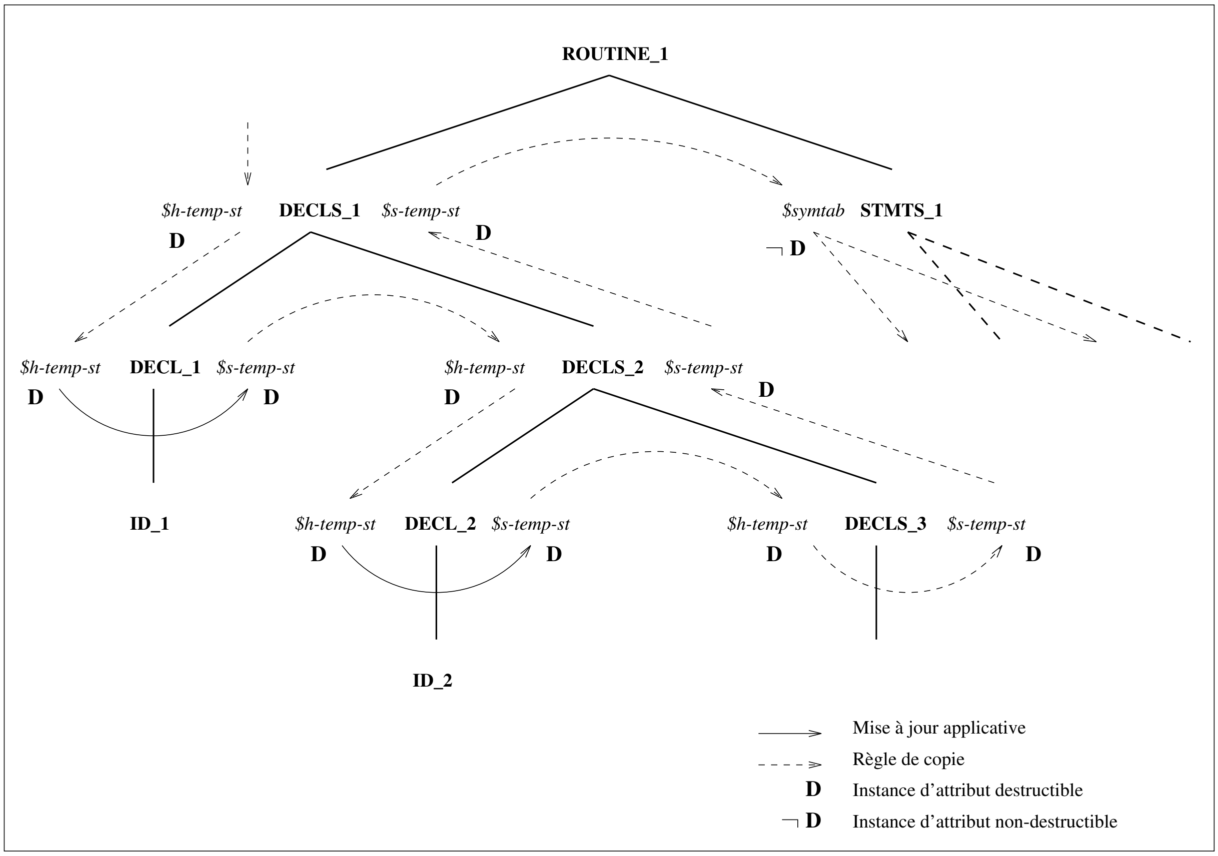


FIG. B.2 - *Destructibilité des instances d'attributs.*

La version optimisée de FNC-2 pourrait donc dans ce cas effectuer toutes les insertions dans la table des symboles de façon destructive, en remplaçant les appels à la fonction de mise à jour *insert* par des appels à l'évaluateur destructif que nous avons obtenu dans l'annexe A.

Notons au passage que l'attribut `$symtab` n'est pas destructible ; en effet, dans la partie “statements”, il est hérité dans chaque branche, et donc partagé. Dans notre langage ROUTINESIMPLE qui n'est pas structuré par blocs, ce n'est pas gênant pour la mise en place de la version destructive de *insert*. Nous allons voir dans l'annexe C que dans un langage structuré par bloc, ce partage entrave bien des optimisations par nos algorithmes actuels.

Annexe C

Contre-exemple

Les deux annexes précédentes nous ont permis de mettre en évidence les optimisations possibles grâce à nos algorithmes. Cependant, comme nous l'avons précisé dans l'annexe B, nous utilisions un langage (RoutineSimple) qui n'était pas structuré par bloc. Même si FORTRAN 77 fait partie de ceux-là, la plupart des langages actuels sont structurés par bloc, pour permettre la déclaration de procédures ou de fonctions et pour avoir les notions de variables globales/variables locales.

Le problème est qu'avec un langage structuré par bloc, on ne peut plus représenter la table des symboles par une structure homogène telle que nous l'avons fait pour RoutineSimple où c'était un arbre binaire de recherche. Il faut différencier chaque bloc dans la structure de la table des symboles, par exemple avec une liste d'arbre binaires où chaque élément de la liste est l'arbre binaire représentant la table des symboles du bloc courant. C'est le cas du langage Simproc du manuel utilisateur d'Fnc-2 [JP 89]. Une nouvelle fonction est donc nécessaire sur le type *symbol-table*:

- *enter-block* prend en entrée une table des symboles et lui ajoute un bloc vide correspondant au nouveau bloc courant (d'un point de vue implantation, elle ajoute un arbre binaire de recherche vide au début de la liste d'arbres binaires).

Nous donnons ici la syntaxe abstraite des arbres représentant un programme écrit en Simproc. C'est l'équivalent du `routine-simple.asx` de l'annexe B. Nous voulons simplement faire apparaître les différences engendrée par la structure de bloc, en particulier pour ce qui est de la déclaration d'une procédure (nous avons étoilé en commentaire les productions primordiales pour notre contre-exemple).

```
{ syntaxe abstraite de simproc: programme simproc-base.asx }

grammar simproc-base is

   root is PROGRAM;

   { programme et blocs }

   PROGRAM     =  program;
   program     -> BLOCK;                                      {****}
   BLOCK       =  block;                                      {****}
   block       -> DECLS STMT;                                 {****}

   { declarations }

   DECLS       =  decls;
   decls       -> DECL*;
   DECL        =  int-decl array-decl proc-decl;              {****}
   int-decl    -> ID  { identificateur de variable }
   array-decl  -> ID  { identificateur de tableau }
                  NUMBER { la dimension };
   proc-decl   -> ID  { identificateur de procedure }         {****}
                  ID  { valeur du parametre }                 {****}
                  ID  { reference du parametre }              {****}
                  BLOCK;                                      {****}

   { statements }

   STMT        =  stmt-list assign call ifthenelse;
   stmt-list   -> STMT+;
   assign      -> VAR EXPR;
   call        -> ID   { identificateur de procedure }
                  EXPR { valeur du parametre }
                  VAR  { reference du parametre };
   ifthenelse  -> EXPR EXPR  { la seule comparaison est l'egalite }
                  STMT { partie true } STMT { partie false };

   { expressions }

   EXPR        =  bin-expr constant VAR { inclusion de phylum };
   bin-expr    -> OP EXPR EXPR;
   constant    -> NUMBER;
   OP          =  plus minus mul div;
   plus        -> ;
   minus       -> ;
   mul         -> ;
   div         -> ;

   { variables }

   VAR         =  simple-var indexed-var;
   simple-var  -> ID  { identificateur de variable }
   indexed-var -> ID  { identificateur de tableau }  EXPR  { index }

   { terminaux }

   NUMBER      =  number;
   number      -> ;
   ID          =  id;
   id          -> ;

end grammar; {simproc-base}
```

La vérification sémantique dans Simproc

Il serait fastidieux de donner ici la grammaire complète qui effectue la phase de vérification sémantique de Simproc (elle est donnée dans [JP 89]). Nous pouvons par contre remarquer que cette phase se déroule différemment de celle de RoutineSimple.

Elle utilise un attribut `$symtab` qui est global, c'est à dire qu'il est attaché à tous les non-terminaux et qu'il est hérité. Il représente constamment la table des symboles générale et subit un *enter-block* à chaque nouvelle déclaration de bloc par une procédure.

Plutôt que de donner la description formelle de cette grammaire attribuée nous allons montrer un exemple d'arbre décoré par les attributs concernant la construction et la propagation de la table des symboles. Pour plus de clarté, nous avons éludé toutes les parties concernant les "statements". La figure C.2 présente une partie d'un exemple où sont déclarés successivement un entier, une procédure et un tableau ; la procédure contient la déclaration d'un entier.

Si nous voulons effectuer des remplacements de fonctions de mise à jour applicatives par leurs versions destructive, il faut s'assurer de la destructibilité de leurs arguments d'entrée.

Tant qu'il s'agit des instances d'appel à des fonctions de mise à jour du premier bloc, nous voyons que les attributs `$h-st` sont destructibles : dans notre exemple, le remplacement de *insert* par sa version destructive peut avoir lieu pour insérer les trois déclarations du premier bloc.

La deuxième déclaration (celle de la procédure) crée dans la table des symboles un nouveau bloc. Celui-ci est ajouté à la version définitive de la table des symboles du bloc courant, qui est véhiculée par l'attribut `$symtab`. Or ce dernier est partagé, puisque l'argument de *enter-block* est la table des symboles du bloc englobant, qui continue à exister pour d'autres déclarations, indépendamment du bloc courant. `$symtab` est donc non destructible. Il en va de même pour l'attribut `$h-st` du non-terminal `BLOC` de cette procédure (puisqu'il est défini à partir de `$symtab`).

A partir du deuxième bloc, on ne peut donc plus effectuer le remplacement d'*insert* par sa version destructive (cf. figure C.1).

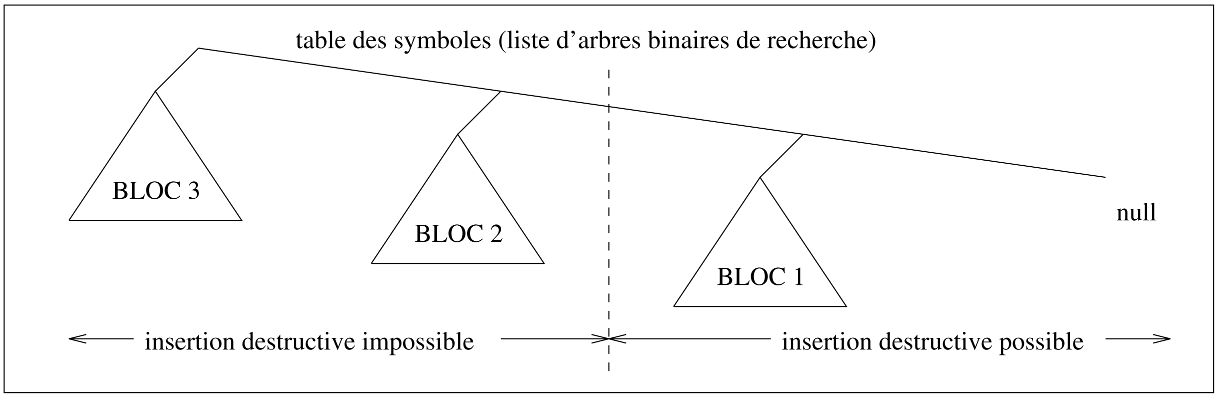


FIG. C.1 - *Le remplacement n'est possible que dans le premier bloc.*

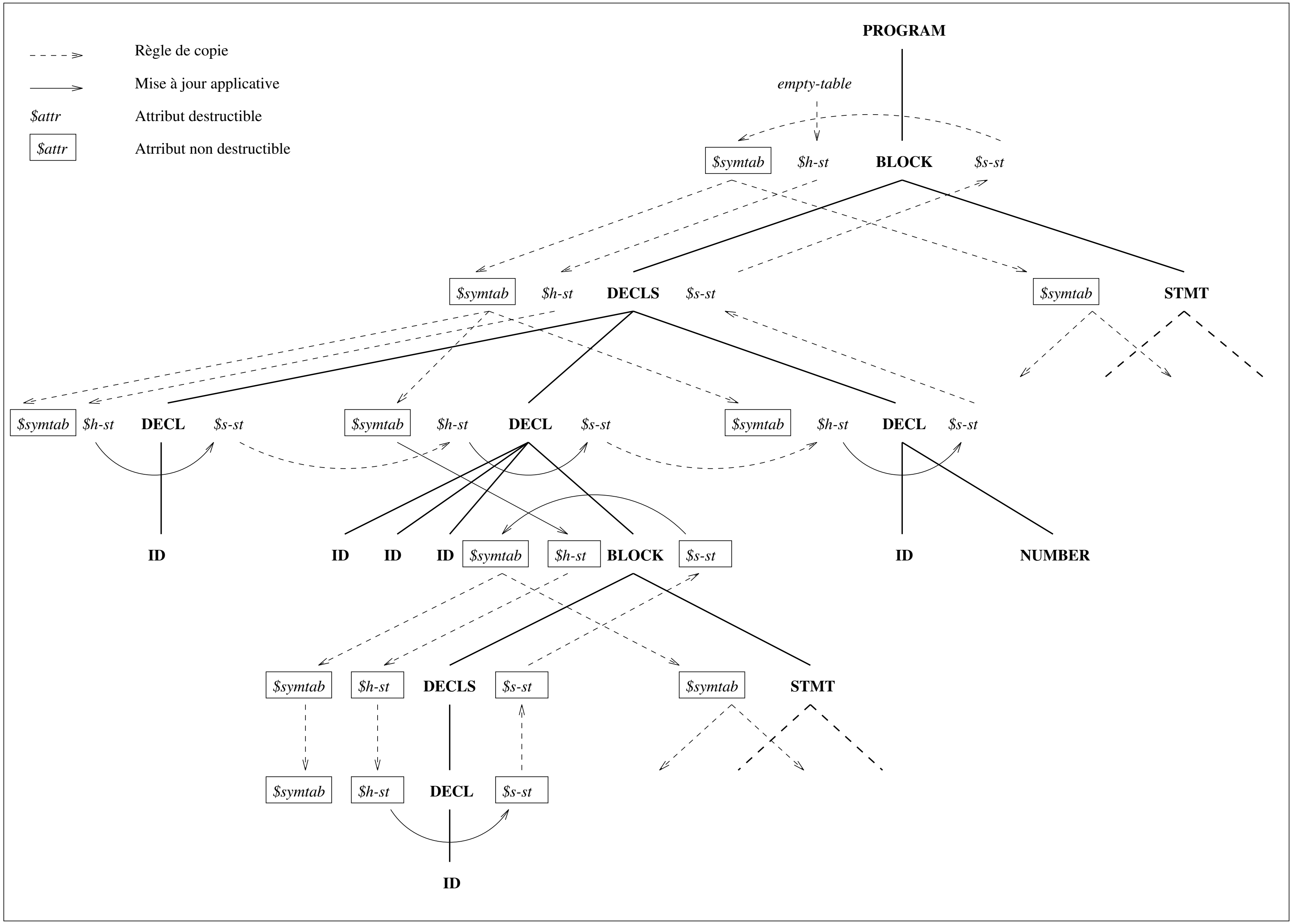


FIG. C.2 - *Destructibilité des instances d'attributs.*

Possibilité d'amélioration

Nous savons que la version destructive de *insert* n'effectue des des modifications physiques **que** sur l'arbre de recherche correspondant au dernier bloc déclaré. Or le partage que prend en compte notre algorithme pour rendre l'attribut `$h-st` non-destructible vise à éviter que l'on détruise physiquement la partie de la table des symboles correspondant aux anciens blocs.

Il y a ici des notions que "nous" prenons en compte, parce que nous connaissons les structures des données et l'effet des fonctions de mise à jour. Nos algorithmes n'ont pas, dans l'état actuel des choses, la possibilité de savoir quelle partie d'une structure de donnée une fonction destructive va modifier physiquement et quel est la teneur exacte du partage entre deux attributs.

Si tel était le cas, nous pourrions effectuer dans cet exemple des mises à jour destructives sur les insertions dans la table des symboles quelle que soit la profondeur du bloc.

Ce problème que nous venons de soulever ouvre la porte à des travaux qui pourraient utiliser les informations dont dispose le système FNC-2 pour affiner la notion de dépendance, en particulier en prenant en compte les origines (structurelles) du partage des attributs.

Bibliographie

[Bl 89] Adrienne Bloss. “Update Analysis and the Efficient Implementation of Functional Aggregates.” *Conf. on Func. Prog. Languages and Computer Architecture.”* London. September 1989.

[Bo 76] Gregor V. Bochmann. “Semantic Evaluation from Left to Right.” *Comm. of ACM.* February 1976.

[BD 77] R. M. Burstall & J. Darlington. “A Transformation System for Developing Recursive Programs.” *Journal of the ACM.* January 1977.

[CC 77] Patrick & Radhia Cousot. “Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximate fixpoints.” *The 4th ACM Symp. on Principles Of Programming Languages.* Los Angeles, California. January 1977.

[DJL 84] Pierre Deransart, Martin Jourdan & Bernard Lorho. “Speeding up Circularity Tests for Attributed Grammars. *Acta Inform.* December 1984.

[DJL 88] Pierre Deransart, Martin Jourdan & Bernard Lorho. “Attribute Grammars : Definitions, Systems and Bibliography.” *Lect. Notes in Comp. Sci., Springer-Verlag.* August 1988.

[De 90] Alain Deutsch. “On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications.” *Les écrits d'ICSLA.* September 1990.

[De 92] Alain Deutsch. “A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence relations.” *Proc. of the IEEE International Conference on Computer Languages.* San Francisco. April 1992.

[De 94] Alain Deutsch. “Interprocedural May-Alias Analysis for Pointers : Beyond k-limiting.” *SIGPLAN Notices, PLDI.* Orlando, Florida. 1990.

[En 84] Joost Engelfriet. “Attribute Grammars : Attribute Evaluation Methods.” *Methods and Tools for Compiler Construction. Bernard Lorho, ed., Cambridge Univ. Press.* Cambridge, 1984.

[GG 84] Harald Ganzinger & Robert Giegerich. “Attribute Coupled Grammars.” *ACM SIGPLAN Notices.* June 1984.

[He 88] Lucy Hederman. "Compile Time Garbage Collection Using Reference Count Analysis." *Rice University.* Houston, Texas. 1988.

[Hu 86] Paul Hudak. "A semantic model of reference counting and its abstraction." *Proc. of the ACM Symp. on LISP and Functional Programming.* August 1986.

[JLM 89] Simon B. Jones & Daniel Le Métayer. "Compile-time garbage collection by sharing analysis." *Conf. on Func. Prog. Languages and Computer Architecture."* London. September 1989.

[JP 89] Martin Jourdan & Didier Parigot. "The FNC-2 System User's Guide and Reference Manual." *INRIA, Rocquencourt.* February 1989 (This manual is periodically updated).

[Ju 86] Catherine Julié. "Optimisation de l'espace mémoire pour les compilateurs générés selon la méthode d'évaluation OAG : étude des travaux de Kastens et propositions d'améliorations." *Rapport de stage DEA d'Orléans.* Septembre 1986.

[Ka 80] Uwe Kastens. Ordered Attribute Grammars. In *Acta Inform. 13.* 1980.

[LH 88] James R. Larus & Paul N. Hilfinger. "Detecting Conflicts Between Structure Accesses." *Proc. of the SIGPLAN Conf. on PLDI.* Atlanta, Georgia. June 1988.

[Pa 88] Didier Parigot. "Transformation, évaluation incrémentale et optimisation des grammaires attribuées : le système FNC-2." *Univ. Paris XI, Centre d'Orsay, nouvelle thèse.* Mai 1988.

[Ro 94] Gilles Roussel. "Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées." *Univ. Paris VI, Thèse de* 3^{eme} *cycle.* Mars 1994.

[RPJ 95] Gilles Roussel, Didier Parigot & Martin Jourdan. "Static and Dynamic Coupling Attribute Evaluators (Extended Abstract)." *Submit for 22nd ACM Symp. on Principles Of Programming Languages.* San Francisco, California. January 1995.

[Se 89] Peter Sestoft. "Replacing Function Parameters by Global Variables." *Conf. on Func. Prog. Languages and Computers Architectures."* London 1989.

[Wa 84] Philip Wadler. "Listless is Better than Laziness. Lazy evaluation and garbage collection at compile-time." *Proc. of the ACM Symp. on LISP and Functional Programming.* Austin, Texas. August 1984.

[Wa 85] Philip Wadler. "Listless is Better than Laziness II. Composing Listless Functions." *Proc. of a workshop on Programs as Data Objects.* Copenhagen, Denmark. October 1985.

[Wa 88] Philip Wadler. "Deforestation : Transforming programs to eliminate trees." *European Symposium On Programming (ESOP).* Nancy, France. 1988.