Stéphane Leibovitsch
Stephane.Leibovitsch@inria.fr

DEA Sémantique, Preuves et Programmation

Université Paris VII

Année universitaire 1995–1996

Rapport de Stage de DEA

Relations entre la sémantique dénotationnelle et les grammaires attribuées

Septembre 1996

Directeur de stage : Didier Parigot

Laboratoire d'accueil : Projet OSCAR — INRIA-Rocquencourt

Table des matières

Table des figures

Liste des algorithmes

Introduction

Il y a trente ans, Knuth [Knu68] introduisait les grammaires attribuées. Depuis elles ont été largement étudiées. Une grammaire attribuée est une spécification déclarative qui décrit comment des attributs (variables) sont calculés sur chaque règle d'une syntaxe particulière. À l'origine, les grammaires attribuées ont été introduites comme un formalisme permettant de spécifier des applications liées à la compilation ; elles étaient destinées à décrire comment décorer un arbre syntaxique et ne pouvaient être utilisées sans une structure sous-jacente (arbre) représentant un programme à traiter. Dans ce type d'applications, deux qualités importantes sont reconnues aux grammaires attribuées :

- elles permettent très naturellement une *décomposition structurelle* qui correspond à la structure syntaxique du langage ;
- elles sont *déclaratives*, dans le sens où l'écrivain spécifie uniquement les règles à utiliser pour calculer la valeur des attributs, et jamais dans quel ordre doivent être effectués ces calculs.

La question importante sur ce formalisme fut alors : "Est-il possible de produire un code exécutable à partir d'une spécification par Grammaires Attribuées ?" La plupart des recherches dans ce domaine se sont donc focalisées sur la production automatique d'un code efficace, et de très bon résultats ont été obtenus, en particulier des techniques d'implémentation efficaces et des méthodes d'analyse statique globale généralement très précises.

La sémantique dénotationnelle est un formalisme dans lequel on associe une fonction sémantique à chacune des productions d'une grammaire hors-contexte, qui permet d'associer une valeur sémantique à une règle syntaxique. Son avantage est de proposer un cadre dans lequel il est plus facile de spécifier précisément la sémantique d'un langage et de faire des preuves sur ce langage.

La combinaison de ces deux formalismes permettrait d'allier la puissance formelle de la sémantique dénotationnelle avec la puissance et l'efficacité des grammaires attribuées.

Plusieurs études sur les relations entre la sémantique dénotationnelle et les grammaires attribuées ont été faites au cours des 15 dernières années, en particulier dans [Mad80, May81, Gan80] mais je me suis basé sur les travaux de Ganzinger [Gan80] car il est le seul à présenter une méthode assez explicite permettant de transformer une sémantique dénotationnelle en grammaire attribuée.

Cependant les grammaires attribuées générées par la méthode de Ganzinger dépassent le formalisme original introduit par Knuth en ce sens qu'elles sont circulaires et nécessite une évaluation paresseuse des attributs [1].

1. Ces notions sont définies dans les premiers chapitres.

L'équipe du projet Oscar a introduit récemment les grammaires attribuées dynamiques [PDRJ95, PRJD96] qui augmentent fortement le pouvoir d'expression des grammaires attribuées, qui deviennent équivalentes aux langages fonctionnels du premier ordre. Les grammaires attribuées dynamiques apportent deux extensions au formalisme de Knuth :

- les *productions conditionnelles* qui permettent de diriger le flot de contrôle grâce à des valeurs d'attributs ;

- les *schémas de production* qui permettent de décrire un schéma de récursion indépendemment de toute structure physique et/ou permettent une combinaison différente des éléments d'une structure physique (production) donnée.

L'objet de mon stage consiste à étudier l'apport des grammaires attribuées dynamiques sur la relations entre la sémantique dénotationnelle et les grammaires attribuées. Plus précisément, ce travail est une extension des travaux de Ganzinger. J'étudie dans ce document la manière dont les grammaires attribuées dynamiques permettent de répondre aux contraintes qu'il impose.

Une étape ultérieure consiste en l'implémentation d'un prototype de traducteur de sémantique dénotationnelle vers les grammaires attribuées en utilisant le système FNC-2 développé dans le projet Oscar qui implémente les grammaires attribuées dynamiques.

Ce document est divisé en trois chapitres. Dans le premier, j'introduis de manière informelle les grammaires attribuées classiques et dynamiques à l'aide d'exemples simples. Dans le second, je présente l'article de Ganzinger et l'apport des grammaires attribuées dynamiques sur ses résultats, ainsi qu'un complément à sa méthode pour la génération automatique d'une grammaire attribuée, et le dernier chapitre est consacré à la présentation du système FNC-2 développé par l'équipe du projet Oscar et l'implémentation que j'ai réalisée.

Chapitre 1

Les grammaires attribuées

Les grammaires attribuées sont un formalisme qui étend les grammaires hors-contexte en leur ajoutant des attributs et des règles sémantiques définissant la valeur de ces attributs. La spécification d'une grammaire attribuée contient uniquement ces règles sémantiques et leur ordre d'évaluation n'est pas indiqué. C'est lors de la compilation qu'il sera calculé.

Les attributs sont divisés en deux catégories selon qu'ils dépendent des attributs associés aux fils d'un nœud ou bien des attributs associés au père d'un nœud. Les premiers sont appelés *attributs synthétisés* et les seconds *attributs hérités*. Les attributs synthétisés représentent un flux de données ascendant alors que les attributs hérités représentent un flux de données descendant.

La première utilisation des grammaires attribuées qui vient à l'esprit est la réalisation de compilateurs, mais leur champ d'application ne s'arrête pas là et les grammaires attribuées peuvent être utilisées pour toutes les manipulations d'arbres. Les grammaires attribuées sont utilisées en particulier pour le traitement des langues naturelles.

1.1 Les grammaires attribuées classiques

Définition 1.1 (Grammaire hors-contexte)
Une grammaire hors-contexte est un tuple $G = (\mathcal{N}, \mathcal{T}, Z, \mathcal{P})$ où

- $\mathcal{N}$ *est un ensemble de non-terminaux,*
- $\mathcal{T}$ *est un ensemble de terminaux,*
- $Z \in \mathcal{N}$ *est un non-terminal particulier appelé* axiome, *et*
- $\mathcal{P}$ *est un ensemble de productions.*

Une production $p \in \mathcal{P}$ est notée $X_0 \to X_1 \cdots X_n, n \geq 0$ où $X_0 \in \mathcal{N}$ et $X_i \in \mathcal{N} \cup \mathcal{T}$.

Un arbre de dérivation est un arbre dont la racine est étiquetée par Z et dont chaque nœud interne est étiqueté par un non-terminal et chaque feuille est étiquetée par un terminal. Un nœud étiqueté par un non-terminal X_0 aura pour fils des nœuds étiquetés par $X_1, \ldots, X_n$ si et seulement si $\exists X_0 \to X_1 \cdots X_n \in \mathcal{P}$.

Le langage $\mathcal{L}(G)$ généré par la grammaire G est l'ensemble des mots pour lesquels il existe un arbre de dérivation.

Définition 1.2 (Grammaire attribuée)
Une grammaire attribuée est un tuple $AG = (G, \mathcal{A}, \mathcal{F})$ *où*

- $G = (\mathcal{N}, \mathcal{T}, Z, \mathcal{P})$ *est une grammaire hors-contexte,*
- $\mathcal{A} = \bigcup \mathcal{A}(X)$ *est un ensemble d'attributs associés à* $X \in \mathcal{N}$, $\mathcal{A}(X) = \mathcal{H}(X) \uplus \mathcal{S}(X)$ *où* $\mathcal{H}(X)$ *est l'ensemble des attributs hérités associés à* X *et* $\mathcal{S}(X)$ *est l'ensemble des attributs synthétisés associés à* X,
- $\mathcal{F} = \bigcup \mathcal{F}(p)$ *est un ensemble de règles sémantiques, où* f_{p,a,X_i} *désigne la règle sémantique définissant l'occurence d'attribut* $a(X_i)$ *dans la production* $p = X_0 \rightarrow X_1 \cdots X_n$ *et* $a \in \mathcal{A}(X_i)$.

Les deux schémas 1.1 et 1.2 expliquent la différence entre les attributs synthétisés et les attributs hérités.

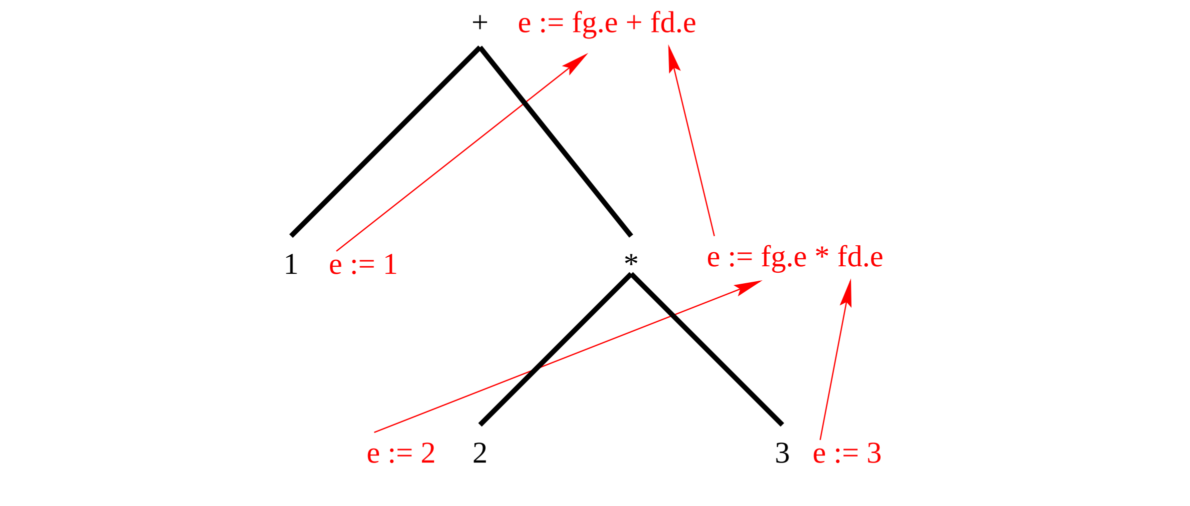

FIG. **1.1** – Attributs synthétisés pour le calcul d'une expression arithmétique

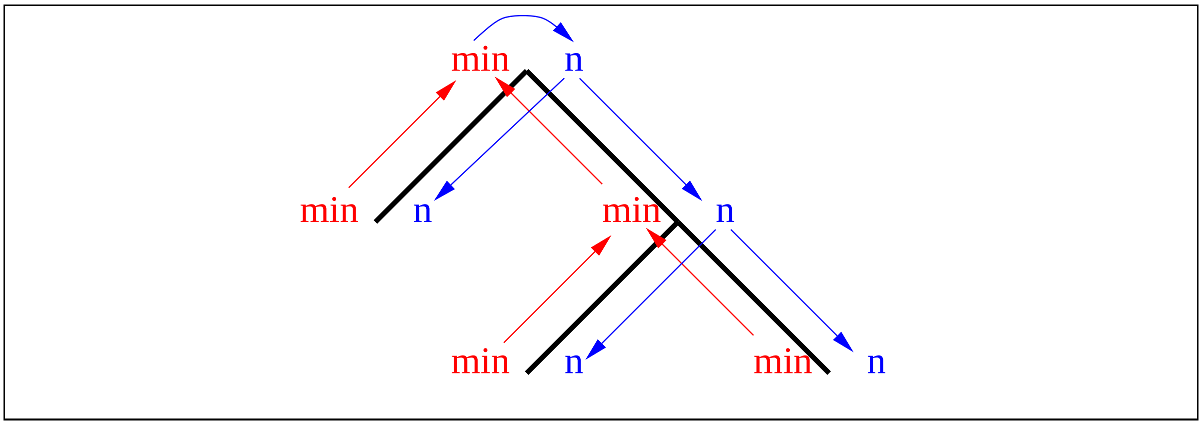

FIG. **1.2** – Attributs hérités : bird

Exemple : les attributs hérités et la grammaire *bird* (Figure 1.2)

Cette grammaire prend un arbre dont toutes les feuilles contiennent une valeur entière. La grammaire attribuée calcule la valeur minimale de toutes ces feuilles grâce à l'attribut synthétisé *min* et stocke à chaque feuille cette valeur dans l'attribut hérité *n*. *min* est calculé à chaque nœud à partir de la valeur sur ses fils tandis que *n* est calculé à chaque nœud à partir de la valeur sur leur père.

Cet exemple montre aussi que l'évaluation de cette grammaire se fait en deux visites de l'arbre : lors de la première, la valeur minimale de toutes les feuilles est calculée et lors de la seconde, on reporte cette valeur sur toutes les feuilles del'arbre.

Le générateur d'évaluateurs de grammaires attribuées doit savoir dans quel ordre évaluer les règles sémantiques pour optimiser le nombre de visites. □

Le manque d'expressivité des grammaires attribuées a conduit à de nombreux travaux consistant à étendre leur formalisme original, par exemple, les grammaires attribuées d'ordre supérieur, les grammaires attribuées orientées objets et les grammaires attribuées circulaires. L'équipe du projet Oscar a proposé deux extensions qui étendent l'expressivité des grammaires attribuées à celle des langages fonctionnels du premier ordre.

Un survol de ces différents types de grammaires attribuées se trouve dans [Paa95].

1.2 Les grammaires attribuées dynamiques

Nous décrivons maintenant les deux extensions que nous utilisons par la suite. Une présentation informelle se trouve dans [PDRJ95] et une description formelle se trouve dans [PRJD96].

Le but de ces deux extensions est d'augmenter le pouvoir d'expression des grammaires attribuées en leur permettant de manipuler des structures non arborescentes, en particulier des structures récursives, et d'orienter le choix d'un ensemble de règles sémantiques non seulement en fonction de la syntaxe (la forme de l'arbre) mais aussi en fonction de la sémantique (la valeur des attributs).

1.2.1 Les productions conditionnelles

Cette extension permet d'associer à chacune des productions plusieurs ensembles de règles sémantiques. Le choix de l'ensemble à utiliser pour calculer les attributs se fait grâce à des conditions que l'on associe à la production.

Les conditions peuvent porter sur tous les attributs d'entrée des non-terminaux de la production.

Les productions conditionnelles peuvent être utilisées dans deux cas :

- choisir entre deux ensembles de règles sémantiques pour une même production ;
- choisir entre plusieurs productions dynamiques (dont la définition se trouve dans la section suivante) associées à une production concrète.

Ces deux cas ne sont pas mutuellement exclusifs. En effet, il est tout à fait possible d'avoir plusieurs ensembles de règles sémantiques pour la même production dynamique.

Cependant, les productions conditionnelles prennent tout leur sens lorsqu'elles sont combinées avec les schémas de production.

1.2.2 Les schémas de production

Cette extension permet d'associer un ensemble de productions virtuelles (ou dynamiques) à une production réelle (ou concrète). Plus précisément, chaque non-terminal d'une production dynamique est associé à un non-terminal de la production concrète (y compris le non-terminal de la partie gauche de la production).

Plusieurs non-terminaux de la production dynamique peuvent faire référence au même non-terminal de la production concrète et tous les non-terminaux de la production concrète ne sont pas nécéssairement référencés dans la production dynamique.

Les schémas de production permettent de décrire des schémas récursifs, indépendemment de la structure de l'arbre.

Ces schémas de production, accompagnés des productions virtuelles, peuvent être utilisés pour manipuler un arbre de syntaxe abstraite virtuel créé dynamiquement lors de l'évaluation de la grammaire attribuée.

Exemple : Quelques possibilités de productions dynamiques Nous allons maintenant présenter différentes utilisations des schémas de production. Nous allons partir d'une production concrète de base :

$$A \rightarrow B\ C\ D$$

dont l'arbre se trouve dans la figure 1.3, à partir de laquelle nous allons créer trois productions dynamiques, mettant ainsi en évidence les trois utilisations possibles.

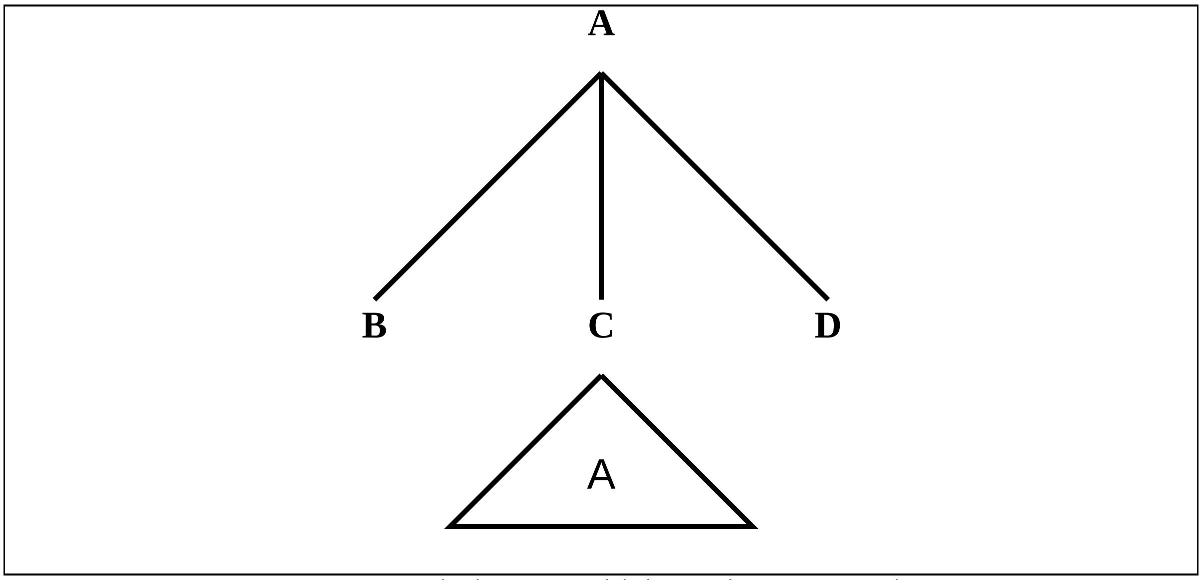

FIG. **1.3** – L'arbre associé à la production concrète

Une première utilisation des schémas de production consiste à cacher des non-terminaux de la partie droite. Dans l'exemple ci-dessous, nous cachons le non-terminal **C**. Le schéma utilisé est :

with scheme
  pas-de-C $\rightarrow B = B\ \ D = D$
end scheme ;

et l'arbre virtuel produit se trouve dans la figure 1.4.

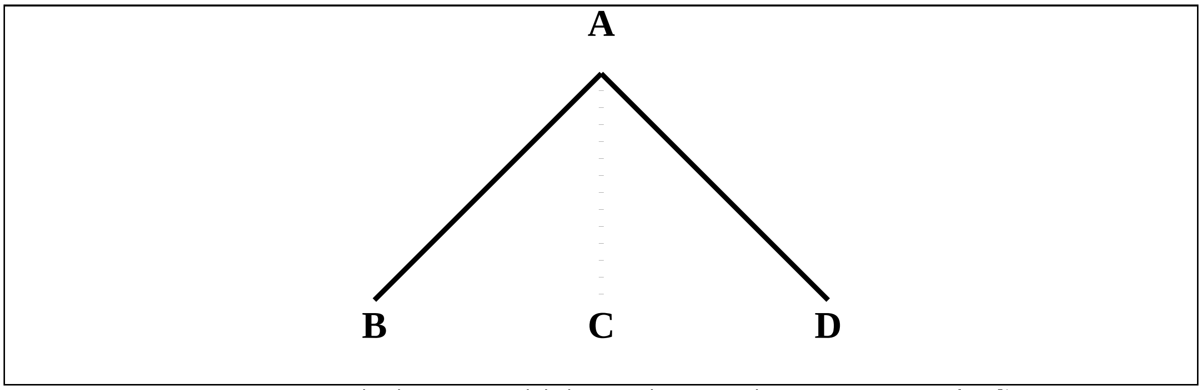

FIG. **1.4** – L'arbre associé à la production dynamique *pas-de-C*

Un second cas dans lequel les schémas de production peuvent être utilisés est celui dans lequel on veut évaluer deux fois les attributs d'un non-terminal de la partie droite de la production concrète avec des attributs hérités différents. Dans ce cas, on crée deux non-terminaux virtuels qui sont tous les deux associés au même non-terminal concret :

with scheme
deux-fois-C $\rightarrow B = B \;\; C_1 = C \;\; C_2 = C \;\; D = D$
end scheme ;

L'arbre virtuel produit par ce schéma se trouve dans la figure 1.5. On remarquera que le sous-arbre dont la racine est **C** dans la production concrète a lui aussi été «dupliqué».

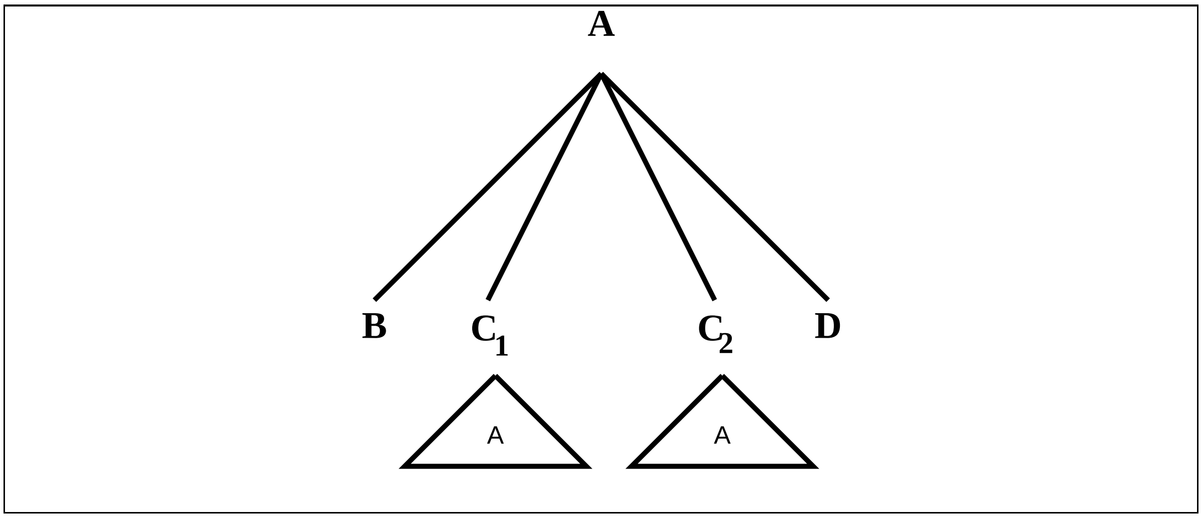

FIG. **1.5** – L'arbre associé à la production dynamique *deux-fois-C*

Le dernier cas que nous présentons ici est celui des schémas récursifs. Ici, l'un des non-terminaux de la production dynamique fait référence au non-terminal de la partie gauche de la production concrète. Cela a pour effet de dupliquer la production , comme cela est montré sur la figure 1.6. Il faut bien sûr ajouter au schéma une production dynamique permettant la terminaison de la récursion, car dans le cas contraire, le sous-arbre sera dupliqué à l'infini.

with scheme
  schema-rec $\rightarrow B = B \;\; C = C \;\; D = D \;\; E = A$
  schema-term $\rightarrow B = B \;\; C = C \;\; D = D$
end scheme ;

.

La figure 1.6 représente un arbre virtuel correspondant à la production dynamique *schema-rec* et le sous-arbre dont la racine est **E** correspond à la production dynamique *schema-term*.

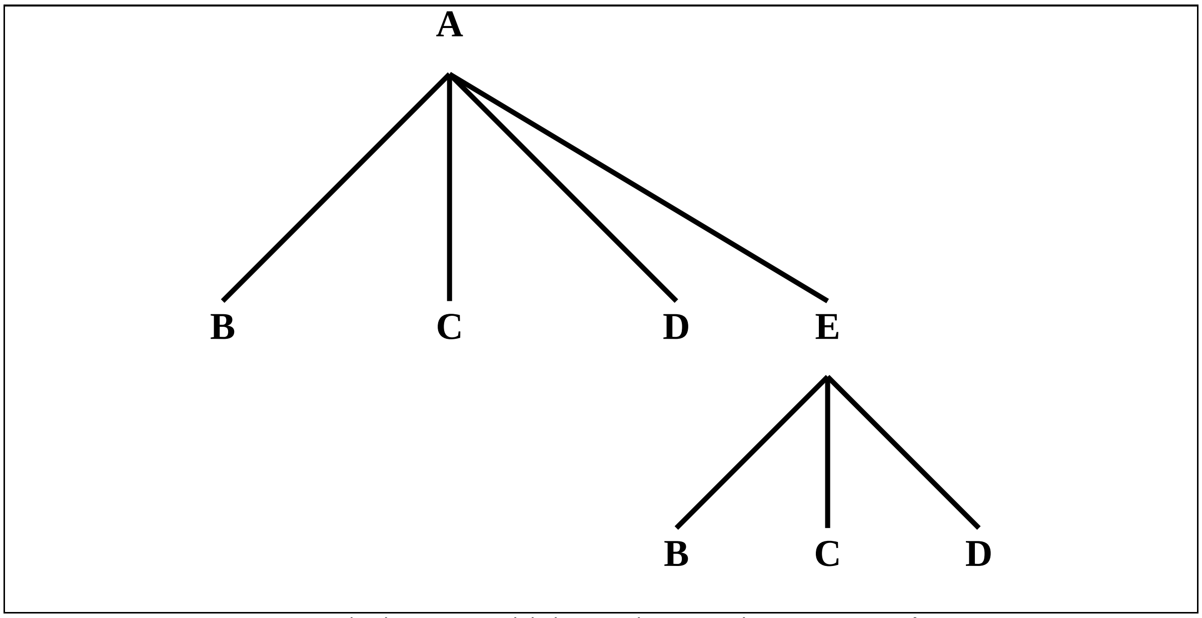

FIG. **1.6** – L'arbre associé à la production dynamique *schema-rec*

Il est bien sûr possible de combiner toutes ces possibilités.

Lorsqu'un schéma de production contient plusieurs productions dynamiques, la sélection de la production dynamique à utiliser se fait à l'aide des productions conditionnelles.

□

Afin de simplifier le formalisme et les algorithmes, nous avons décidé de travailler sur des syntaxes abstraites qui sont une forme plus simple de grammaires hors-contexte.

Syntaxe abstraite : Une syntaxe abstraite est une syntaxe dans laquelle aucun terminal n'apparaît. Une syntaxe abstraite diffère d'une syntaxe concrète par le fait qu'elle n'est pas soumise aux contraintes de l'analyse syntaxique, en ce qui concerne la gestion des priorités des opérateurs par exemple, ce qui la rend beaucoup plus simple et compacte. C'est sur de telles syntaxes que nous travaillerons.

Les syntaxes abstraites sont définies à partir de *phyla* (sing. phylum), qui correspondent à des types en ML, et d'*opérateurs* qui correspondent à des constructeurs en ML.

Exemple : les expressions arithmétiques La grammaire hors-contexte :

$$\begin{array}{lcl} E & \rightarrow & E+E \\ E & \rightarrow & E*E \\ E & \rightarrow & E/E \\ E & \rightarrow & E-E \\ E & \rightarrow & (E) \\ E & \rightarrow & I \end{array}$$

La syntaxe abstraite sous forme de phyla et d'opérateurs :

```
     E   =   plus moins mult divis val
  plus   ->  E E
 moins   ->  E E
  mult   ->  E E
 divis   ->  E E
   val   ->
```

Ici, `E` est un phylum et `plus`, `moins`, `mult` et `divis` sont les opérateurs.
Le type ML :

type E =
    plus **of** E * E
| moins **of** E * E
| mult **of** E * E
| divis **of** E * E
| val

Les valeurs effectives seront représentées par des attributs des feuilles. □

Les syntaxes abstraites sont définies de manière plus formelle dans [KLMM83].

Chapitre 2

Transformation d'une sémantique dénotationnelle en Grammaire Attribuée

Ce chapitre décrit en détail la transformation d'une sémantique dénotationnelle en grammaire attribuée. Après avoir donné quelques définitions et discuté de l'évaluation paresseuse dans les grammaires attribuées, nous expliquons la méthode que Ganzinger a utilisé dans son article, expliquons les problèmes rencontrés et donnons les solutions apportées par les grammaires attribuées dynamiques. Une dernière partie est consacrée à l'algorithme global effectuant cette transformation.

Note: Tous les exemples de cette partie sont tirés de l'article de Ganzinger et sont basés sur son formalisme.

Note: Les algorithmes et les types sont donnés dans une syntaxe proche de celle de Caml Light.

2.1 Définitions et notations

Une *fonction sémantique* est une fonction qui associe un objet syntaxique à un objet sémantique. Les valeurs que peut prendre une telle fonction appartiennent à un *domaine sémantique* qui a pour avantage sur les ensembles classiques de permettre l'application du théorème de Tarski sur le calcul du plus petit point fixe d'un système d'équations. Nous ne l'utilisons cependant pas ici.

Une *sémantique dénotationnelle* est un ensemble de fonctions sémantiques. Chacune d'elle est associée à une production d'une grammaire hors-contexte et permet d'associer à toute expression du langage $\mathcal{L}(G)$ une valeur sémantique.

Définition 2.1 (Domaine)
Les domaines sont des treillis complets.

Le produit cartésien de domaines est noté $D = D_1 \times \cdots \times D_n$. *Une notation plus complète est :* $D = [s_1 : D_1; \cdots; s_n : D_n]$ *si le nom du sélecteur est utile.* s_i *est le nom du* i^{e} *sélecteur.*

Le domaine $D = D_1 + \cdots + D_n$ *est défini par* $D = \{(i, x) \mid 1 \leq i \leq n, x \in D_i\} \cup \{\perp, \top\}$.

*La notation x **to** D_i représente le fait que $x_i \in D_i$, c'est à dire que (i, x_i) **to** $D_i = x_i$.*
*Réciproquement, si $x_i \in D_i$, alors x_i **in** $D = (i, x_i)$.*
Si D_1 et D_2 sont des domaines, alors $D = D_1 \rightarrow D_2$ est le domaine des fonctions continues de D_1 dans D_2.

Notation : La i^{e} projection d'un produit de domaines est notée D^i.

Définition 2.2 (Expression)
Une expression du meta-langage décrivant une sémantique dénotationnelle a la syntaxe BNF suivante :
*E:= int | bool | **let** ids = E **in** E | **if** E **then** E **else** E | $[\![id]\!](ids)$ | ids*
ids:= id , ids | id
où $[\![id]\!](ids)$ est une référence à une fonction sémantique (voir la définition suivante) et les autres constructions ont leur sens habituel.

Définition 2.3 (Sémantique dénotationnelle)
Une sémantique dénotationnelle basée sur une grammaire hors-contexte $G = (\mathcal{T}, \mathcal{N}, Z, \mathcal{P})$ est un objet où chaque élément de $\mathcal{N}$ est associé à un domaine et chaque élément $p : X_0 \rightarrow X_1 \cdots X_n$ de $\mathcal{P}$ est associé à une λ-expression dont le domaine doit correspondre au domaine de X_0 de la forme $\lambda a \cdot E$ où E est une expression dont la définition est donnée ci-dessus.
*Cette λ-expression sera appelée **fonction sémantique** dans la suite du texte.*
Les paramètres d'entrée de la fonction sémantique associée au non-terminal Z correspondent aux paramètres d'entrée du programme.
La sémantique d'une expression du langage $\mathcal{L}(G)$ est la valeur de la fonction sémantique associée au non-terminal Z, la racine de l'arbre de dérivation de l'expression.

Définition 2.4 (Dépendances des attributs et circularité)
1. *Un attribut a dépend d'un attribut b si la règle sémantique définissant a contient b.*
2. *Le graphe de dépendance des attributs est un graphe dirigé où les sommets sont les attributs, et deux attributs sont reliés par une arête si l'un dépend de l'autre.*
3. *Si le graphe de dépendance des attributs est cyclique, alors les dépendances sont circulaires.*

Notation : L'expression **letrec** $x_1 = a_1, \ldots, x_n = a_n$ **in** *body* est une abbréviation pour :

$$(\lambda x_1, \ldots, x_n \cdot body)(\mathbf{fix}(\lambda x_1, \ldots, x_n \cdot (a_1, \ldots, a_n)))$$

où **fix** est l'opérateur de point fixe.

2.2 Evaluation paresseuse des attributs

2.2.1 Le problème

L'évaluation paresseuse des attributs telle que l'entend Ganzinger correspond en fait à l'évaluation par nécessité des règles sémantiques. Cela est particulièrement visible dans le cas de la conditionnelle. En effet, une évaluation stricte de la conditionnelle conduit à un cycle dans le graphe de dépendances des attributs. Cela est détaillé dans l'exemple suivant.

Boyland décrit dans l'introduction de [Boy96] le problème posé par l'évaluation stricte des expressions conditionnelles dans les grammaires attribuées. Il montre bien que cela conduit à des grammaires attribuées circulaires.

Exemple : La circularité dûe à l'évaluation stricte de l'expression conditionnelle Cet exemple est tiré de [Boy96].

Voici un exemple de grammaire attribuée qui va induire un cycle dans le graphe de dépendances de la figure 2.1. Cette grammaire attribuée contient trois productions et on associe deux attributs au non-terminal A :

- l'attribut synthétisé s ;
- l'attribut hérité i.

Le problème est posé par le fait que l'expression "**if** e_1 **then** e_2 **else** e_3" est évaluée comme un simple appel de fonction **IfThenElse**(e_1, e_2, e_3). On va donc créer des dépendances entre d'une part e_2 et e_3, et d'autre part l'attribut pour lequel on est en train de définir la règle sémantique.

where $S \rightarrow A$ **use**
    \$i(A) := 0
    \$val(S) := \$s(A)
end where ;

where $A_0 \rightarrow\ A_1\ A_2$ **use**
    \$s(A_0) := **if** odd(\$i(A_0)) **then** \$s(A_1) **else** \$s(A_2) ;
    \$i(A_1) := **if** odd(\$i(A_0)) **then** \$s(A_2) **else** \$i(A_0) + 1 ;
    \$i(A_2) := **if** odd(\$i(A_0)) **then** \$i(A_0) + 1 **else** \$s(A_1) ;
end where ;

where $A \rightarrow$ **use**
    \$s(A) := $\$i(A) + 1$;
end where ;

□

2.2.2 La solution adoptée dans ce travail

Nous décrivons tout d'abord le résultat que nous voulons obtenir et nous montrons ensuite rapidement comment y arriver. Cela sera détaillé dans la section 2.6.

On voit très clairement le problème dans le cas de la conditionnelle (fig. 2.2). En effet, le nœud de la condition a 3 fils : la condition, le *then* et le *else*. Nous voulons évaluer les attributs du sous-arbre *then* seulement dans le cas où la condition est vérifiée. De même, nous voulons évaluer les attributs du sous-arbre *else* seulement dans le cas où si la condition est fausse.

La figure 2.2 représente les deux sous-arbres qu'il est possible d'évaluer[1]. Le sous-arbre de gauche représente le cas où l'on veut évaluer la branche *then* et le sous-arbre de droite

1. 'Évaluer un sous-arbre' est un abus de langage et signifie : 'évaluer les attributs de ce sous-arbre'

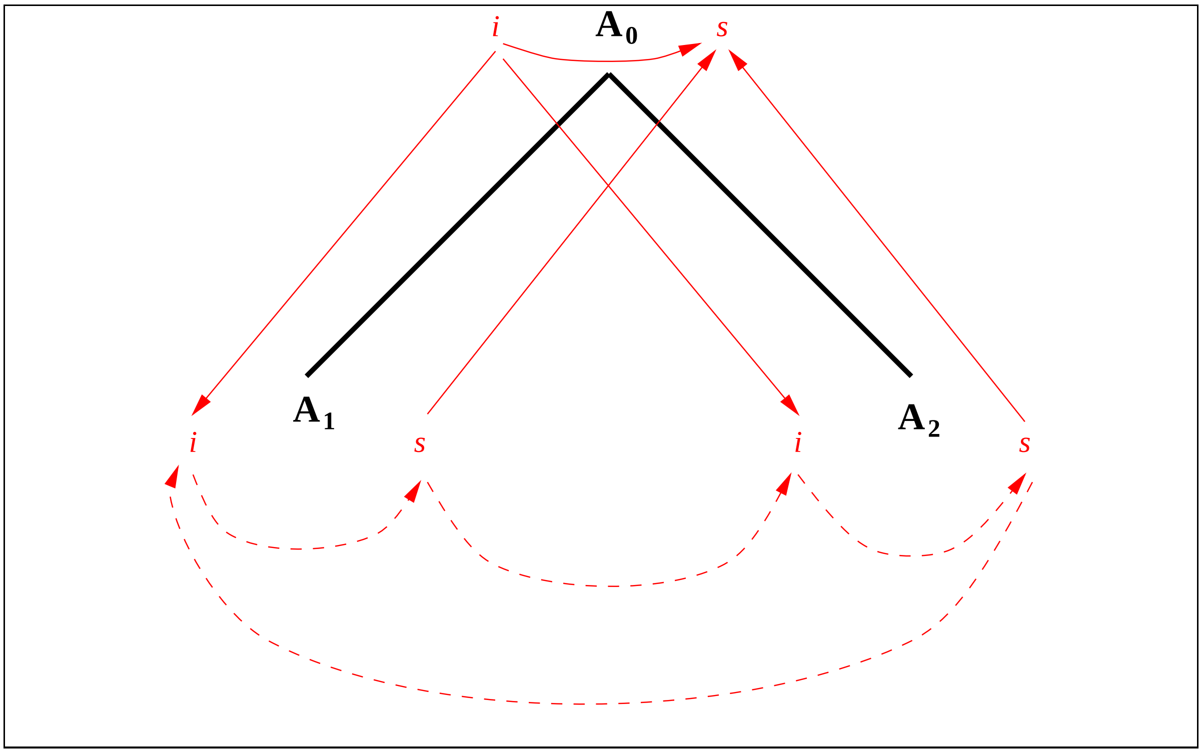

FIG. **2.1** – Graphe de dépendance de la grammaire attribuée de Boyland

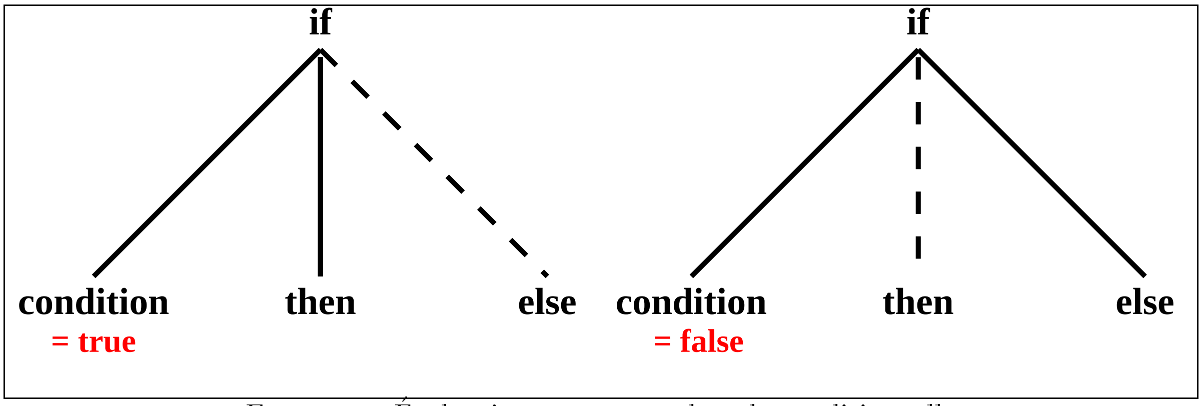

FIG. **2.2** – Évaluation paresseuse dans la conditionnelle représente le cas où l'on veut évaluer la branche *else*. Les branches en pointillés représentent les sous-arbres que l'on ne veut pas évaluer.

Cet exemple nous montre bien comment les schémas de production peuvent intervenir pour "cacher" les sous-arbres que nous ne voulons pas évaluer, et comment les productions conditionnelles peuvent être utilisées pour sélectionner le schéma à utiliser.

2.3 L'article de Ganzinger

Après avoir introduit la forme grammaire attribuée décrite dans l'article de Ganzinger, nous donnons les différentes étapes de la transformation d'une sémantique dénotationnelle en grammaire attribuée, et nous terminons cette section en donnant un exemple complet de cette transformation.

Dans son article, Ganzinger cherche à construire automatiquement un compilateur à partir d'une sémantique dénotationnelle. Il procède pour cela en trois étapes :

- génération d'une forme intermédiaire à partir de la sémantique dénotationnelle ;
- séparation entre la sémantique statique et la sémantique dynamique, puis génération d'une grammaire attribuée et d'une interprétation de l'arbre syntaxique d'entrée ;
- transformation de l'interprétation en une description d'une génération de code (cette étape est juste mentionnée dans l'article).

Ganzinger définit les attributs statiques comme les attributs ne dépendant pas des attributs hérités de l'axiome et les attributs dynamiques comme les attributs dépendant de l'axiome. Les sémantiques statique et dynamique sont calculées à partir de cette classification des attributs.

Plus intuitivement, les attributs statiques peuvent être calculés au moment de la compilation alors que les attributs dynamiques ne peuvent être calculés qu'au moment de l'exécution. Les attributs hérités de l'axiome correspondent aux paramètres d'entrée du programme.

La grammaire attribuée générée par cette méthode nécessite que les attributs soient évalués paresseusement et que les dépendances circulaires entre les attributs soient possibles. Les cas qui font apparaître ces contraintes seront détaillés dans la section 2.4.

L'objet du stage est d'étudier la manière dont les grammaires attribuées dynamiques permettent de remédier à ces contraintes.

Ce que nous cherchons à faire est de générer un interpréteur pour la sémantique dénotationnelle d'entrée. C'est pourquoi seule la première étape décrite par Ganzinger nous intéresse. En effet, il n'y a pas de séparation entre la sémantique statique et la sémantique dynamique dans notre cas car nous cherchons à réaliser un interpréteur plutôt qu'un compilateur. Cette étape est divisée par Ganzinger en quatre phases distinctes.

L'objet obtenu est toujours une fonction sémantique mais sa forme est beaucoup plus adaptée à la transformation en grammaire attribuée que Ganzinger appelle "attribute-grammar form" (forme grammaire attribuée) définie ci-dessous. Je vais ensuite décrire les différentes phases de la transformation.

2.3.1 La forme grammaire attribuée

Une forme grammaire attribuée est toujours une fonction sémantique mais elle a l'avantage d'être syntaxiquement très proche d'une grammaire attribuée. Il devient donc ensuite très facile de transformer cette forme grammaire attribuée en véritable grammaire attribuée.

Une forme grammaire attribuée peut être assimilée à un système d'équations de point fixe avec deux types d'équations :

- les équations qui font référence aux fonctions sémantiques associées aux non-terminaux de la partie droite de la production, que Ganzinger appelle équations d'entrée/sortie (I/O equations), et
- les équations qui définissent la valeur des paramètres d'entrée des non-terminaux de la partie droite de la production et les valeurs des paramètres de sortie du non-terminal de la partie gauche de la production. Ce sont elles qui effectuent les véritables calculs. Ganzinger les appelle équations sémantiques.

Voici une définition formelle de la forme grammaire attribuée, tirée de l'article de Ganzinger.

Note : Nous supposons que les grammaires attribuées avec lesquelles nous travaillons sont basées sur des syntaxes abstraites et que les productions ne contiennent donc que des non-terminaux. Cela nous permettra de simplifier la définition originale de Ganzinger.

Définition 2.5 (Forme grammaire attribuée)
Une fonction sémantique associée à une production $p = X_0 \rightarrow X_1 \cdots X_n$ est dite en forme grammaire attribuée si et seulement si :

1. *son domaine sémantique est du premier ordre, c'est-à-dire un domaine fonctionnel avec les paramètres d'entrée et de sortie formant deux tuples. Il est de la forme :*

$$D[X_0] = [(a_1 : D^1_{in}[X_0], \ldots, a_m : D^m_{in}[X_0]) \rightarrow (b_1 : D^1_o[X_0], \ldots, b_m : D^k_o[X_0])]$$

   Les a_i sont les paramètres d'entrée de $[\![X_0]\!]$ et les b_i sont ses paramètres de sortie.

2. *elle est de la forme :*

$$\begin{array}{l} [\![X_0]\!] = \lambda a_1, \ldots, a_{m_o}. \\ \quad \textbf{\textit{letrec}} \\ \quad\quad (y_{11}, \ldots, y_{1k_1}) = [\![X_1]\!](x_{11}, \ldots, x_{1m_1}), \\ \quad\quad (y_{l1}, \ldots, y_{lk_n}) = [\![X_l]\!](x_{l1}, \ldots, x_{lm_l}), \\ \quad\quad z_1 = F_1, \ldots, z_L = F_L \\ \quad \textbf{in } (b_1, \ldots, b_{k_o}) \end{array}$$

   où

   (a) *les fonctions sémantiques $[\![X_i]\!]$ ont le domaine $D[X_i]$,*

   (b) *les $x_{ij}, 1 \leq i \leq l, 1 \leq j \leq m_i$, et les $y_{ij}, 1 \leq i \leq l, 1 \leq j \leq m_i$ sont des variables deux à deux distinctes sur les domaines $D^j_{in}[X_i]$ et $D^j_o[X_i]$, respectivement. Nous supposons que $m_i = m_{X_i}$ et que $k_i = k_{X_i}$, le nombre de paramètres d'entrée et de sortie de X_i. On dit que x_{ij} denote le j^e paramètre d'entrée de X_i et que y_{ij} denote le j^e paramètre de sortie de X_i. Ces équations sont les équations d'entrée/sortie.*

(c) *les variables* $z_1, \ldots, z_L$ *constituent le tuple de tous les paramètres d'entrée des* $[\![X_i]\!], x \geq 1$ *et des paramètres de sortie de* $[\![X_0]\!]$.*Les équations définissants ces variables sont les équations sémantiques.*

(d) *les* F_μ *sont des termes arbitraires pouvant contenir les variables* x_{ij}, y_{ij}, a_i, b_i *et toutes les fonctions sémantiques* $[\![X_i]\!], 0 \leq i \leq n$.

Le premier point de la définition permet d'associer un attribut à chacun des paramètres de la fonction. En effet, les attributs sont des valeurs du premier ordre, et c'est pour cette raison que l'on effectue cette transformation. Nous cherchons à tranformer une fonction type fonctionnel classique, qui prend un paramètre et qui renvoie une fonction qui prend un paramètre, etc...en une fonction qui prend n paramètres, éventuellement d'ordre supérieur.

Le second point spécifie la forme que doit prendre la fonction sémantique. Comme on peut le constater cette forme est très proche d'un ensemble de règles sémantiques associées à une production :

- les équations d'entrées/sorties correspondent au flot de données entre les nœuds de l'arbre syntaxique ; elles correspondent aux règles de copies dans une grammaire attribuée ;
- les équations sémantiques définissent les règles sémantiques qui effectuent les véritables calculs.

La correspondance avec les grammaires attribuées est la suivante :

1. les attributs hérités a_i de domaine $D^i_{in}[X_0], 1 \leq i \leq m_{X_0}$ et les attributs synthétisés b_i de domaine $D^i_o[X_0], 1 \leq i \leq m_{X_0}$ sont attachés au non-terminal X.
2. les variables a_i, b_i, x_{ij}, y_{ij} sont des noms pour les attributs $a_i[X_0], b_i[X_0], a_i[X_j]$ et $b_i[X_j]$, respectivement. La règle sémantique $z_j = F_j$ est la règle pour l'attribut dénoté par z_j.

Nous décrivons maintenant les différentes étapes à appliquer à une fonction sémantique pour la transformer en forme grammaire attribuée, telles qu'elles sont données dans l'article de Ganzinger. Nous donnons ensuite un exemple complet, étape par étape de cette transformation.

2.3.2 Transformation des domaines sémantiques

Cette étape consiste à tenter éliminer l'ordre supérieur des domaines des fonctions sémantiques. Pour cela, un domaine de la forme

$$D_0 \to \cdots \to D_n$$

sera transformé en un domaine de la forme

$$D_0 \times \cdots \times D_{n-1} \to D_n.$$

Le but de cette étape est d'obtenir des fonctions sémantiques du premier ordre afin de se rapprocher d'une forme grammaire attribuée et de faire ressortir clairement les paramètres d'entrée et de sortie de la fonction. Les paramètres d'entrée de la fonction peuvent cependant être d'ordre supérieur.

2.3.3 Génération des équations pour les paramètres d'entrée

Cette étape consiste à générer pour chaque appel à une fonction sémantique les équations associant les paramètres formels de cette fonction à ses paramètres effectifs. On génère ainsi le flux descendant des paramètres.

Pour être plus précis, chaque fonction sémantique est associée à une production de la grammaire du langage qu'il faut transformer en un ensemble d'attributs et de leurs règles de calcul. Les paramètres d'entrée de la fonction deviennent des attributs hérités alors que ses paramètres de sortie deviennent des attributs synthétisés.

Ganzinger donne très peu de détails quant à la réalisation de cette phase, mais c'est elle qui nécessite les grammaires attribuées étendues, comme nous le verrons dans la section 2.4.

2.3.4 Génération des équations de sortie

Cette étape consiste à générer pour chacun des paramètres de sortie de la fonction une équation associant une variable à chaque expression de sortie de la fonction. Je considère que la fonction a plusieurs expressions de sortie si elle renvoie un tuple ou un enregistrement (voir section 2.5.2).

2.3.5 Simplification du système d'équations

Cette étape consiste à simplifier les règles pour les paramètres de sortie en appliquant les autres équations et les règles de transformations, telles que la β-réduction. Le but de cette étape est d'éliminer le plus possible les appels aux fonctions sémantiques.

Là aussi, Ganzinger donne peu de détails sur le processus exact à appliquer.

2.3.6 Génération de la grammaire attribuée

Cette étape intervient après la séparation entre la sémantique statique et la sémantique dynamique dans l'article de Ganzinger, mais je vais la décrire maintenant car les étapes intermédiaires ne nous intéressent pas ici.

Il s'agit de générer les règles sémantiques définitives pour la grammaire attribuée. La sémantique dénotationnelle que nous avons obtenue ici est déjà dans une forme très proche de la grammaire attribuée que nous souhaitons obtenir ; il faut en fait transformer les variables en attributs, et leur associer un non-terminal. La manière dont chaque attribut est associé à un non-terminal est détaillée dans l'article de Ganzinger.

- tous les paramètres d'entrée de la fonction sémantique deviennent des attributs hérités ;
- tous les paramètres de sortie de la fonction sémantique deviennent des attributs synthétisés ;
- le non-terminal associé à chaque attribut hérité est celui dont l'attribut apparaît comme paramètre ;
- le non-terminal associé à chaque attribut synthétisé est celui apparaissant en partie gauche de la production associée à la fonction sémantique de départ ;

– les équations de la forme : $(b_1, \ldots, b_n) = [\![X]\!](a_1, \ldots, a_m)$ sont remplacées par les règles sémantiques :

$$\begin{array}{rcl} \$b_1 & := & \$o_1(X) \\ & \vdots & \\ \$b_n & := & \$o_n(X) \end{array}$$

où $o_1, \ldots, o_n$ sont les n attributs de sortie (synthétisés) du non-terminal X.

Il est à noter que la grammaire attribuée générée ainsi a de fortes chances d'être circulaire, c'est à dire d'avoir un graphe de dépendance des attributs contenant des cycles.

Exemple : le séquencement Voici un exemple de grammaire attribuée générée pour le séquencement. On obtient une grammaire attribuée non circulaire et ne posant pas de problème.

On associe la règle sémantique

$$\begin{array}{l} [\![stlist_0]\!] = \lambda env \cdot \lambda s \cdot \\ \qquad [\![stlist_1]\!](env, [\![stat]\!](env, s)) \end{array}$$

à la production :

$$stlist_0 \longrightarrow stat \,;\, stlist_1$$

Le domaine des non-terminaux *stlist* et *stat* est : $D_{env} \rightarrow D_{state} \rightarrow D_{state}$.

Cette règle évalue d'abord l'instruction *stat*, qui renvoie un nouvel état du programme, puis évalue le reste de la séquence dans le nouvel état.

La première étape consistant à défonctionnaliser le domaine va générer le domaine $D_{env} \times D_{state} \rightarrow D_{state}$ et la fonction sémantique va devenir :

$$\begin{array}{l} [\![stlist_0]\!] = \lambda env, s \cdot \\ \qquad [\![stlist_1]\!](env, [\![stat]\!](env, s)) \end{array}$$

La seconde étape consistant à générer le flux descendant des paramètres d'entrée de la fonction sémantique va générer les équations :

$$\begin{array}{rcl} env_1 & = & env \\ s_1 & = & s \\ env_2 & = & env \\ s_2 & = & [\![stat]\!](env_1, s_1) \end{array}$$

La troisième étape consistant à générer l'équation de sortie de la fonction sémantique va générer l'équation :

$$o \;=\; [\![stlist_1]\!](env, [\![stat]\!](env, s))$$

Enfin, la dernière étape consistant à simplifier l'équation de sortie va générer l'équation :

$$o \quad = \quad [\![stlist_1]\!](env_2, s_2)$$

La sémantique dénotationnelle sous forme grammaire attribuée est donc :

$$\begin{array}{l}
[\![stlist_0]\!] = \lambda env, s\cdot \\
\quad \textbf{letrec} \\
\qquad o = [\![stlist_1]\!](env_2, s_2), \\
\qquad env_1 = env, \\
\qquad s_1 = s, \\
\qquad env_2 = env, \\
\qquad s_2 = [\![stat]\!](env_1, s_1) \\
\quad \textbf{in } o
\end{array}$$

où les équations d'entrées/sorties sont celles qui définissent o et s_2 et les équations sémantiques sont celles qui définissent env_1, s_1 et env_2.

$$\begin{array}{rcl|l}
\$o(stlist_0) & := & \$o(stlist_1) & \text{La valeur de sortie de la } \lambda\text{-expression} \\
\$env(stat) & := & \$env(stlist_0) & \$env(stat) \Longleftrightarrow env_1 \\
\$s(stat) & := & \$s(stlist_0) & \$s(stat) \Longleftrightarrow s_1 \\
\$env(stlist_1) & := & \$env(stlist_0) & \$env(stlist_1) \Longleftrightarrow env_2 \\
\$s(stlist_1) & := & \$o(stat) & \$s(stlist_1) \Longleftrightarrow s_2
\end{array}$$

La figure 2.3 représente les dépendances des attributs sur l'arbre syntaxique correspondant à cette règle.

□

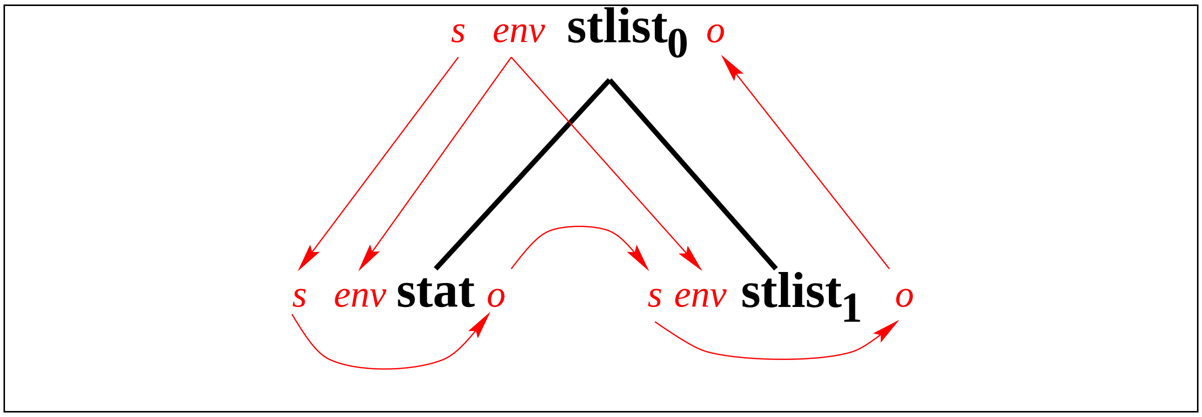

FIG. **2.3** – Graphe de dépendances correspondant à la séquence

2.4 Les contraintes sur la grammaire attribuée générée

Nous caractérisons dans cette section les différents problèmes posés par la transformation de Ganzinger, décrite dans la section précédente, et donnons des algorithmes permettant de détecter les cas dans la sémantique dénotationnelle conduisant à ces problèmes lors de la transformation. Nous donnons dans chaque cas un exemple de fonction sémantique qui met le problème en lumière.

Définition 2.6 (Types de données)
Voici les types de données que nous utilisons dans les algorithmes des sections suivantes. Le type expression *correspond aux expressions E du meta-langage utilisé pour définir des sémantiques dénotationnelles (voir definition 2.2) et le type* equation *correspond aux équations générées par le traducteur.*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{\textit{type}}\ \textit{expression} =} \\
 & \textit{Id} & \textbf{\textit{of}}\ \textit{string} \\
| & \textit{Int} & \textbf{\textit{of}}\ \textit{int} \\
| & \textit{NonTerm} & \textbf{\textit{of}}\ \textit{string*(expression list)} \\
| & \textit{IfThenElse} & \textbf{\textit{of}}\ \textit{expression*expression*expression} \\
| & \textit{Let} & \textbf{\textit{of}}\ \textit{string*expression*expression} \\
| & \textit{Tuple} & \textbf{\textit{of}}\ \textit{expression list}
\end{array}
$$

$$\textbf{\textit{type}}\ equation = \{partie\_gauche : string; partie\_droite : expression\}$$

Les problèmes liés aux grammaires attribuées générées par cette méthode sont de deux natures :

- les problèmes liés à la récursivité, qui vont générer des dépendances circulaires des attributs ;
- les problèmes liés à l'évaluation paresseuse, qui vont nécessiter l'évaluation paresseuse des attributs.

Il y a un troisième problème qui correspond à la présence dans une expression de deux appels distincts à la même fonction sémantique avec des paramètres différents.

Ces trois problèmes correspondent en fait à tous les cas particuliers que l'on peut rencontrer.

2.4.1 L'appel récursif à la fonction sémantique

Soit une production

$$X_0 \longrightarrow X_1 \cdots X_n$$

Si la fonction sémantique associée à cette production fait un appel récursif à $[\![X_0]\!]$, nous obtiendrons, lors de la transformation en grammaire attribuée, une définition de l'attribut de sortie $\$o(X_0)$ en fonction de lui-même : $\$o(X_0) := f(\ldots, \$o(X_0), \ldots)$.

Cela nécéssite un traitement particulier. Ganzinger considère que les règles sémantiques forment un système d'équations de point fixe qu'il faut résoudre pour calculer la valeur d'un attribut.

Nous pouvons détecter ce problème en faisant une recherche dans l'expression de la fonction de tous les appels à une fonction sémantique. Si l'un de ces appels fait référence à la fonction que nous sommes en train de définir, alors nous sommes bien dans ce cas.

2.4.2 L'appel à deux fonctions sémantiques, en fonction d'une condition

Ce problème intervient sur un nœud qui possède deux fils, dont l'évaluation dépend d'une condition. La grammaire attribuée générée par la méthode exposée précédemment provoque le calcul des attributs synthétisés pour les deux fonctions sémantiques. Cela n'est pas souhaitable.

Voici l'algorithme à utiliser pour détecter ce cas (algorithme 2.1). Il est bien sûr incomplet et il faut lui rajouter toutes les autres constructions pour les autres types d'expressions.

Le type de l'algorithme est : expression → string set*bool.

Dès que l'on se trouve sur un nœud **IfThenElse** dans l'expression à analyser, on regarde si l'ensemble des non-terminaux de la branche **Then** est identique à l'ensemble des non-terminaux de la branche **Else**. Dans ce cas, il n'y a pas de problème mais dans le cas contraire, le problème doit être résolu.

La valeur de retour de l'algorithme est un couple dont le premier élément correspond à l'ensemble des non-terminaux de la fonction et le second est égal à **true** si la fonction présente le problème et **false** sinon.

$$
\begin{array}{lll}
\textbf{let rec}\ \text{test\_if} = \textbf{function} & & \\
\quad \text{NonTerm}(\mathit{id,params}) & \rightarrow & \{\mathit{id}\},\ \textbf{false} \\
\mid\ \ \text{IfThenElse}(c, t, e) & \rightarrow & \textbf{let}\ \mathit{T\_set},\ \mathit{T\_bool} = \text{test\_if}\ t \\
 & & \textbf{and}\ \mathit{E\_set},\ \mathit{E\_bool} = \text{test\_if}\ e\ \textbf{in} \\
 & & \quad \mathit{T\_set} \cup \mathit{E\_set}, (\mathit{T\_set} \neq \mathit{E\_set} \vee \mathit{T\_bool} \vee \mathit{E\_bool})
\end{array}
$$

Algorithme 2.1: Détection du cas de l'appel à deux fonctions sémantiques en fonction d'une condition

Exemple : la boucle *while* Un cas où l'évaluation paresseuse est nécessaire est celui des appels récursifs. En effet une évaluation stricte générerait une boucle infinie car l'appel récursif serait évalué même si la condition de terminaison était vraie.

Les boucles sont traitées par des appels récursifs. Voici la boucle **while**.

La production $stat \rightarrow$ **while** $expr$ **do** $stlist$ est associée avec la fonction sémantique :

$$
\begin{array}{l}
[\![stat]\!] = \lambda env \cdot \lambda s \cdot \\
\quad \textbf{let}\ (v, s') = [\![expr]\!](env, s)\ \textbf{in if}\ v\ \textbf{then} \\
\quad\quad [\![stat]\!](env, [\![stlist]\!](env, s')) \\
\quad \textbf{else} \\
\quad\quad s'
\end{array}
$$

On va transformer cette fonction en l'ensemble de règles sémantiques suivant :

$$
\begin{array}{rcl}
env_1 & = & env \\
s_1 & = & s \\
(v, s') & = & [\![expr]\!](env_1, s_1) \\
env_2 & = & env \\
env_3 & = & env
\end{array}
$$

$$
\begin{aligned}
o_2 &= [\![stlist]\!](env_3, s') \\
o &= \textbf{if } v \textbf{ then } [\![stat]\!](env_2, o_2) \textbf{ else } s'
\end{aligned}
$$

La variable o_2, qui correspond au corps de la boucle, sera évaluée quel que soit le résultat du test. C'est ce que l'on veut éviter. On ne veut évaluer le corps de la boucle que si le test est positif. □

2.4.3 Deux appels à la même fonction sémantique avec des paramètres distincts

Une telle construction générerait une grammaire attribuée avec deux ensembles d'équations correspondant à la fonction sémantique concernée, or cela est totalement interdit car chaque attribut doit être défini par une règle sémantique unique.

La détection de ce cas se fait en parcourant l'arbre syntaxique de l'expression et en recherchant deux appels à la même fonction sémantique, *mais avec des paramètres différents.*

Exemple : l'instruction *double* L'instruction **double** permet d'exécuter deux fois la même instruction.

La production $stat \rightarrow$ **double** $stlist$ est associée à la fonction sémantique :

$$
\begin{aligned}
&[\![stat]\!] = \lambda env \cdot \lambda s \cdot \\
&\qquad [\![stlist]\!](env, [\![stlist]\!](env, s))
\end{aligned}
$$

On voit ici que les deux appels à $[\![stlist]\!]$ ont des paramètres distincts et nous sommes donc bien dans le cas qui va nous poser un problème, comme le montre le système d'équations généré :

$$
\begin{aligned}
env_1 &= env && (2.1) \\
env_2 &= env && (2.2) \\
s_2 &= s && (2.3) \\
o_1 &= [\![stlist]\!](env_2, s_2) && (2.4) \\
o &= [\![stlist]\!](env_1, o_1) && (2.5)
\end{aligned}
$$

On voit ici que les équations 2.4 et 2.5 font toutes deux référence à la même fonction sémantique $[\![stlist]\!]$ avec des paramètres différents. Ce système d'équations, lorsqu'il est transformé en grammaire attribuée, définit deux fois les attributs d'entrée pour $stlist$:

$$
\begin{aligned}
\$env(stlist) &:= \$env(stat) && (2.6) \\
\$env(stlist) &:= \$env(stat) && (2.7) \\
\$s(stlist) &:= \$s(stat) && (2.8) \\
\$s(stlist) &:= \$o(stlist) && (2.9) \\
\$o(stat) &:= \$o(stlist) && (2.10)
\end{aligned}
$$

Les équations 2.6 et 2.7 ne posent pas de problème car il s'agit de la même règle mais les équations 2.8 et 2.9 entrent en conflit l'une avec l'autre. En effet, les deux règles sont différentes. De plus l'équation 2.9 introduit une circularité dans les dépendances des attributs (voir fig. 2.4). Le chemin circulaire est dessiné en pointillés. □

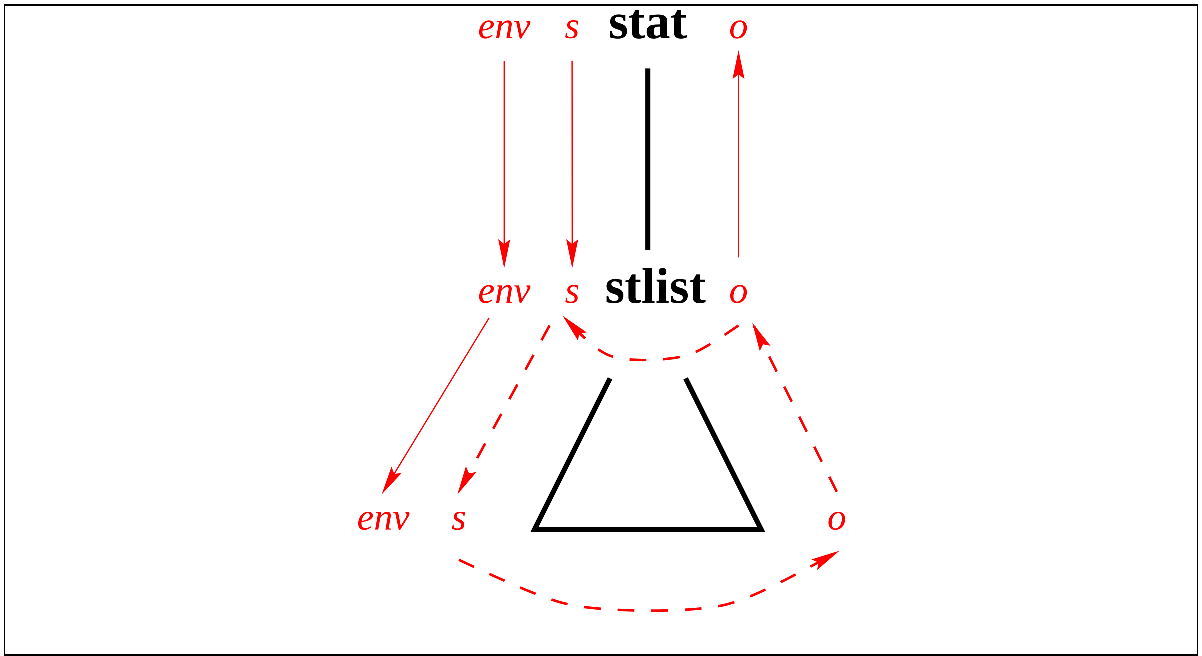

FIG. **2.4** – Circularité dans les dépendances des attributs

2.5 Les algorithmes et les problèmes annexes

La section suivante décrit la manière dont les grammaires attribuées dynamiques permettent de résoudre les problèmes que nous venons d'évoquer mais il existe quelques problèmes annexes qu'il faut résoudre. Il s'agit du calcul des dépendances des variables et des attributs dans une fonction sémantique et une forme grammaire attribuée, ainsi que du problème de l'éclatement, qui consiste à transformer une expression de type n-uplet en n expressions. Le but de cet étape est d'obtenir des attributs indépendants les uns des autres dans la grammaire.

2.5.1 Le calcul des dépendances

Nous avons besoin de calculer deux types de dépendances :

- les dépendances d'une expression par rapport aux variables dont elle a besoin ;
- les dépendances d'une expression par rapport aux non-terminaux dont elle a besoin.

Les deux algorithmes sont très proches l'un de l'autre.

Exemple : les dépendances Voici un système d'équations correspondant au séquencement. Il s'agit ici d'une forme intermédiaire générée par l'algorithme.

$$s_1 \quad := \quad s$$

$$\begin{array}{rcl} env_1 & := & env \\ s' & := & [\![stat]\!](env_1, s_1) \\ env_2 & := & env \\ o & := & [\![stlist_1]\!](env_2, s') \end{array}$$

L'algorithme de dépendances par rapport aux non-terminaux appliqué à o génère la liste suivante : $stlist_1$, $stat$.

L'algorithme de dépendances par rapport aux variables appliqué à o génère la liste suivante : env_2, s', env, env_1, s_1, s.

□

Un objet de ce type représente l'équation $partie\_gauche = partie\_droite$.

Fonctions annexes : Les algorithmes utilisent deux fonctions annexes :

- get_ids: *expression* → *string list* qui renvoie la liste des variables apparaissant dans l'expression passée en paramètre ;
- get_nts: *expression* → *string list* qui renvoie la liste des non-terminaux apparaissant dans l'expression passée en paramètre.

Les dépendances par rapport aux variables

Cette fonction est de type : string → equation list → string set

L'algorithme prend en entrée la variable *var* dont on cherche les dépendances et la liste des équations générées *liste* lors d'une étape précédente. Il renvoie l'ensemble des variables dont dépend *var*.

On cherche dans la liste des équations celle qui donne sa valeur à la variable. On recherche toutes les variables apparaissant dans cette expression, que l'on ajoute à l'ensemble des dépendances, puis on rappelle récursivement l'algorithme avec chacune de ces variables.

L'algorithme termine lorsque aucune équation n'est trouvée pour la variable passée en paramètre.

$$\begin{array}{lll} \textbf{let rec}\ \text{make\_depend}\ var\ liste = \textbf{match}\ liste\ \textbf{with} & & \\ \quad [\,] & \rightarrow & \emptyset \\ | \quad x :: l & \rightarrow & \textbf{if}\ var = x.partie\_gauche\ \textbf{then} \\ & & \quad \textbf{let}\ dep = \text{get\_ids}\ x.partie\_droite\ \textbf{in} \\ & & \quad\quad dep \cup \bigcup_{v \in dep}(\text{make\_depend}\ v\ liste) \\ & & \textbf{else}\ \text{make\_depend}\ var\ l \end{array}$$

Algorithme 2.2: Calcul des dépendances par rapport aux variables

Attention : Cet algorithme suppose qu'il n'y a pas de cycle dans le système d'équations. Dans le cas qui nous intéresse, cela ne peut pas arriver car les algorithmes de traitement des cas particuliers, qui pourraient, eux, générer cette circularité, l'ont éliminé.

Les dépendances par rapport aux non-terminaux

L'algorithme est strictement identique au précédent mais on va chercher la liste des non-terminaux en plus de la liste des variables et ce sont eux que nous ajoutons à l'ensemble des dépendances plutôt que les variables.

Cette fonction est de type :string $\rightarrow$ equation list $\rightarrow$ string set

```
let rec make_depend_nts var liste = match liste with
      []      →  ∅
|     x :: l →  if var = x.partie_gauche then
                     let dep = get_ids x.partie_droite
                     and nts = get_nts x.partie_droite in
                         nts ∪ ⋃_{v∈dep}(make_depend_nts v liste)
                else make_depend_nts var l
```

Algorithme 2.3: Calcul des dépendances par rapport aux non-terminaux

2.5.2 Le problème des tuples et la factorisation

Le problème posé par les tuples existe aussi dans le cas des enregistrements et sera traité de la même manière. Ce sont en effet tous les deux des types composés et c'est ce qui nous intéresse ici.

Nous voulons ici éclater les tuples et les enregistrements, de manière à obtenir plusieurs attributs de type simple, plutôt qu'un attribut d'un type composé. La raison qui nous pousse à faire cela est de garder une certaine indépendance entre les différentes composantes et de se rapprocher le plus possible de l'esprit et du formalisme des grammaires attribuées. Le compilateur pourra alors effectuer une analyse plus fine.

Lorsqu'une fonction sémantique renvoie un tuple (ou un enregistrement), la première étape est d'effectuer une projection de ce tuple sur chacun de ses éléments. Cela génère autant de règles sémantiques quasiment identiques qu'il y a d'éléments dans le tuple. Il s'agit de l'algorithme 2.4.

```
let rec eclate = function
      (Id _ | Int _) as x  →  [x]
|     Let(id, e1, e2)       →  let res = eclate e2 in
                                    map (function x → Let(id, e1, x)) r1
|     IfThenElse(c, e, t)   →  let r1 = eclate t
                               and r2 = eclate e in
                                    map2(function x y → IfThenElse(c, x, y)) r1 r2
|     Tuple(t)              →  t
```

Algorithme 2.4: Éclatement d'un tuple

La seconde étape est de prendre ces règles une par une afin d'effectuer quelques optimisations simples, telles que l'élimination des variables locales non utilisées et l'élimination de conditions dont les deux branches seraient identiques. L'exemple ci-dessous permet de mieux comprendre de quoi il s'agit.

Exemple : la factorisation Soit la production var $\longrightarrow$ **id** [expr] à laquelle on associe la règle sémantique suivante :

$$
\begin{array}{l}
[\![var]\!] = \lambda env \cdot \lambda s \cdot \\
\quad \textbf{if } env(id).ds = variable \textbf{ then} \\
\quad\quad \textbf{let } v = env(id).inf \textbf{ to } VAR, \\
\quad\quad\quad\quad (ind, t_{ind}, s') = [\![exp]\!](env, s) \textbf{ in} \\
\quad\quad\quad \textbf{if } t_{ind} = \text{int} \textbf{ and } v.t = \text{array} \textbf{ then} \\
\quad\quad\quad\quad \textbf{let} \quad l = ((s'.ss(v.addr)) \textbf{ to } ARRAY).lb, \\
\quad\quad\quad\quad\quad\quad u = ((s'.ss(v.addr)) \textbf{ to } ARRAY).ub \textbf{ in} \\
\quad\quad\quad\quad\quad \textbf{if } l \leq (ind \textbf{ to } I) \textbf{ and } (ind \textbf{ to } I) \leq u \textbf{ then} \\
\quad\quad\quad\quad\quad\quad (array\_e\_type, v.addr, ind \textbf{ to } I, s') \\
\quad\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad\quad (array\_e\_type, v.addr, ind, \perp, s') \\
\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad (error\_type, undef\_address, \perp, \perp) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad (error\_type, undef\_address, \perp, \perp)
\end{array}
$$

Cette fonction se charge de renvoyer des informations sur la donnée à laquelle on veut accéder dans une variable tableau. Les informations renvoyées sont :

- le type du tableau $array\_e\_type$ ou bien une erreur $error\_type$,
- l'adresse du tableau ou bien une erreur $undef\_address$,
- l'indice de la valeur que l'on cherche ou bien une erreur $\perp$, et
- le nouvel état du programme s' ou bien une erreur $\perp$.

Cette fonction vérifie d'abord que l'identificateur est bien une variable, puis récupère un ensemble d'informations à son sujet, dans v. Elle va ensuite chercher des informations sur l'indice, qui est une expression. Elle vérifie ensuite que l'indice est bien un entier et que la variable est bien un tableau. Dans ce cas elle vérifie que l'indice est bien dans l'intervalle des bornes du tableau (l et u), et si toutes ces conditions sont vérifiées, renvoie les informations décrites ci-dessus.

Pour la valeur du premier attribut, $array\_e\_type$, on peut éliminer le **if** le plus interne car les deux valeurs renvoyées sont identiques. Cette suppression permet également l'élimination du **let** précédent car les variables locales l et u ne sont plus utilisées. C'est aussi le cas pour les deux derniers attributs.

Cette factorisation génère donc quatre règles sémantiques pour les quatre éléments du tuple dont certaines peuvent être simplifiées.

Voici l'expression générée pour la première règle :

$$
\begin{array}{l}
o_1 := \\
\quad \textbf{if } env(id).ds = variable \textbf{ then} \\
\quad\quad \textbf{let } v = env(id).inf \textbf{ to } VAR, \\
\quad\quad\quad\quad (ind, t_{ind}, s') = [\![exp]\!](env, s) \textbf{ in} \\
\quad\quad\quad \textbf{if } t_{ind} = \mathrm{int} \textbf{ and } v.t = \mathrm{array} \textbf{ then} \\
\quad\quad\quad\quad\quad array\_e\_type \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad error\_type \\
\quad \textbf{else} \\
\quad\quad error\_type
\end{array}
$$

On a ici éliminé le **if** le plus interne (celui qui teste le dépassement de l'intervalle des bornes du tableau), car il n'influe pas sur la valeur de sortie du premier élément du tuple. On a aussi éliminé les variables locales u et v car elles ne sont utilisées que dans le test que l'on a éliminé. □

Nous avons étudié la manière dont Ganzinger transforme une fonction sémantique en grammaire attribuée et les contraintes que cela impose. Nous avons caractérisé les situations qui posent des problèmes et nous avons des algorithmes pour les détecter. Nous allons maintenant étudier de quelle manière les grammaires attribuées dynamiques permettent de résoudre ces problèmes.

2.6 L'apport des grammaires attribuées dynamiques

Nous décrivons dans cette section les méthodes que nous proposons pour résoudre les problèmes décrits dans la section précédente. Après avoir décrit ces différentes méthodes, nous indiquons la manière dont les conditions des productions conditionnelles doivent être générées.

Les trois problèmes cités peuvent se résoudre grâce aux productions conditionnelles et aux schémas de production.

Les schémas de production vont permettre de gérer l'évaluation paresseuse. Il suffira pour cela de ne mettre dans les productions virtuelles que les non-terminaux dont on va vouloir évaluer les attributs.

Les productions conditionnelles vont permettre de diriger le flot de contrôle vers la bonne production virtuelle.

Nous détaillons maintenant de manière précise comment utiliser les schémas de production et les productions conditionnelles pour traiter chacun des trois cas vus précédemment.

2.6.1 L'appel récursif à la fonction sémantique

Tout d'abord, nous supposons que l'appel récursif est à l'intérieur d'une conditionnelle, ce qui n'est pas une contrainte très forte car dans le cas contraire, on obtiendrait une suite infinie d'appels, c'est-à-dire un programme qui boucle.

Soit une production $X_0 \to X_1 \cdots X_n$ à laquelle une fonction sémantique de la forme

$$
\begin{array}{l}
[\![X_0]\!] = \lambda \ldots \cdot \\
\quad \cdots [\![X_i]\!] \cdots [\![X_j]\!] \cdots \\
\quad \textbf{if } \mathit{condition} \textbf{ then} \\
\quad\quad [\![X_0]\!](\cdots [\![X_i]\!] \cdots) \\
\quad \textbf{else} \\
\quad\quad \cdots [\![X_j]\!] \cdots \\
\quad \cdots
\end{array}
$$

Le schéma de production que nous voulons obtenir à partir de cette règle va être :

$$
\begin{array}{l}
\textbf{with scheme} \\
\quad X_0 r \to X_{i_1} \cdots X_{i_k} \; X_r = X_0 \\
\quad X_0 t \to X_{j_1} \cdots X_{j_l} \\
\textbf{end scheme}
\end{array}
$$

Les X_i (resp. les X_j) correspondent aux dépendances de l'appel récursif (resp. de la partie terminale) de l'expression.

Les X_i correspondent à la suite de non-terminaux nécessaires à l'évaluation de la partie récursive. Cela inclut en particulier l'évaluation de *condition* ainsi que le calcul des valeurs de toutes les variables utilisées dans l'appel récursif.

Les X_j correspondent à la suite de non-terminaux nécessaires à l'évaluation de la partie terminale (non récursive). Cela inclut en particulier l'évaluation de *condition* et de *expression* ainsi que toutes les variables utilisées dans *condition* et *expression*.

L'ensemble $X_i \cap X_j$ n'est pas nécessairement vide.

Le X_r est un non-terminal virtuel correspondant à une nouvelle évaluation de la fonction sémantique.

Une fois ce schéma de production créé, il faut modifier dans la fonction sémantique d'origine l'appel récursif pour le transformer en : $[\![X_r]\!]$.

Une dernière étape consiste à reporter la condition dans les productions conditionnelles de la manière suivante :

$$
\begin{array}{l}
\textbf{where } [\mathit{condition}] X_0 r \to X_{i_1} \cdots X_{i_k} \; X_r \;\; \textbf{use} \\
\quad \ldots\ldots \\
\textbf{end where} \\
\textbf{where } [] X_0 t \to X_{j_1} \cdots X_{j_l} \; \textbf{use} \\
\quad \ldots\ldots \\
\textbf{end where}
\end{array}
$$

Note : Il faut faire attention aux conditionnelles imbriquées. Dans ce cas il faut reporter toutes les conditions dans des productions conditionnelles. La manière détaillée de réaliser cela est expliquée dans la section 2.6.4.

Les équations seront ensuite générées normalement puis réparties dans les deux productions conditionnelles décrites ci-dessus.

Exemple : La fonction sémantique associée au *while* Cet exemple fait appel à l'algorithme permettant de traiter les conditionnelles imbriquées. C'est pourquoi il y a trois productions conditionnelles dans le schéma présenté ici. Cet algorithme est détaillé dans la section 2.6.4.

Soit la production

$$\text{stat} \longrightarrow \textbf{while}\ \text{expr}\ \textbf{do}\ \text{stlist}$$

à laquelle on associe la règle sémantique suivante :

$$\begin{array}{l}
[\![\ stat]\!] = \lambda env \cdot \lambda s \cdot \\
\quad \textbf{let}\ (ev, t, s') = [\![expr]\!](env, s)\ \textbf{in} \\
\quad\quad \textbf{if}\ t = bool\ \textbf{then} \\
\quad\quad\quad \textbf{if}\ ev\ \textbf{to}\ B\ \textbf{then} \\
\quad\quad\quad\quad \textbf{let}\ s'' = [\![stlist]\!](env, s')\ \textbf{in} \\
\quad\quad\quad\quad\quad [\![stat]\!](env, s'') \\
\quad\quad\quad \textbf{else}\ s' \\
\quad\quad \textbf{else}\ \perp
\end{array}$$

Le domaine des non-terminaux *stat* et *stlist* est : $D_{env} \rightarrow D_{state} \rightarrow D_{state}$.

Le domaine associé au non-terminal *expr* est : $D_{env} \rightarrow D_{state} \rightarrow (D_{value} \times D_{type} \times D_{state})$.

Le premier **let** permet d'évaluer la condition et le tuple contient la valeur, le type et le nouvel état. On teste ensuite le type et la valeur de la condition et on évalue le corps du **while** avant d'appeler la fonction récursivement.

Le schéma de production suivant est généré :

$$\begin{array}{l}
\text{stat} \longrightarrow \textbf{while}\ \text{expr}\ \textbf{do}\ \text{stlist} \\
\textbf{with scheme} \\
\quad \text{wr} \longrightarrow \text{expr stlist}\ W = \text{wr} \\
\quad \text{wt} \longrightarrow \text{expr} \\
\textbf{end scheme}
\end{array}$$

Les productions conditionnelles générées sont :

$$\begin{array}{l}
\textbf{where}\ [\$ev(wr)\ \textbf{to}\ B \wedge \$t(wr) = bool]\ wr \longrightarrow \text{expr stlist}\ W \\
\textbf{where}\ [\$t(wt) = bool]\ wt \longrightarrow \text{expr} \\
\textbf{where}\ []\text{wt} \longrightarrow \text{expr}
\end{array}$$

Le système d'équations généré pour la première production est :

$$\begin{array}{rcl}
env_1 & := & env \\
s_1 & := & s \\
(ev, t, s') & := & [\![expr]\!](env_1, s_1) \\
env_2 & := & env \\
s'_2 & := & s \\
s'' & := & [\![stlist]\!](env_2, s'_2) \\
env_3 & := & env \\
s''_3 & := & s'' \\
o & := & [\![W]\!](env_3, s''_3)
\end{array}$$

□

2.6.2 L'appel à deux fonctions sémantiques, en fonction d'une condition

Ce cas est traité de manière très similaire au cas précédent. En fait, la seule différence concerne le schéma de production : on n'y ajoute pas de non-terminal virtuel X_r car il n'y a pas d'appel récursif.

Exemple : la conditionnelle On associe à la production

$$\textbf{stat} \longrightarrow \textbf{if } exp \textbf{ then } stlist_1 \textbf{ else } stlist_2 \textbf{ end if}$$

la règle :

$$\begin{array}{l} [\![stat]\!] = \lambda env \cdot \lambda s \cdot \\ \quad \textbf{let } (ev, t, s') = [\![exp]\!](env, s) \textbf{ in} \\ \quad\quad \textbf{if } t = bool \textbf{ then} \\ \quad\quad\quad \textbf{if } ev \textbf{ to } B \textbf{ then} \\ \quad\quad\quad\quad [\![stlist_1]\!](env, s') \\ \quad\quad\quad \textbf{else} \\ \quad\quad\quad\quad [\![stlist_2]\!](env, s') \\ \quad\quad \textbf{else } \bot \end{array}$$

Cette fonction teste le type de la condition et sa valeur booléenne et appelle en fonction de ce test l'une ou l'autre des *stlist*.

Le schéma de production que nous allons associer à cette règle va être :

$$\begin{array}{l} \textbf{with scheme} \\ \quad st\_then \longrightarrow exp\ stlist_1 \\ \quad st\_else \longrightarrow exp\ stlist_2 \\ \quad st\_error \longrightarrow exp \\ \textbf{end scheme} \end{array}$$

La production dynamique pour *st_then* sera utilisée dans le cas où les deux conditions $t = bool$ et ev **to** B sont réunies. La production pour *st_else* sera utilisée dans le cas ou $t = bool$ est vérifiée et ev **to** B est fausse, et la production pour *st_error* sera utilisée dans le cas où les deux conditions sont fausses. Nous étudierons dans la section 2.6.4 comment les conditions sont associées à chaque production dynamique.

□

2.6.3 Deux appels à la même fonction sémantique avec des paramètres distincts

Ce cas est très simple et ne nécessite qu'un schéma de production avec une unique production virtuelle où le non-terminal appelé deux fois est dupliqué :

Soit la production $X_0 \rightarrow X_1 \cdots X_i \cdots X_n$ où X_i est le non-terminal appelé deux fois. Le schéma de production généré va être le suivant :

$$\begin{array}{l} \textbf{with scheme} \\ \quad X_0 \rightarrow X_1 \cdots X_{i1} = X_i\ X_{i2} = X_i \cdots X_n \\ \textbf{end scheme} \end{array}$$

Il faut ensuite remplacer dans la fonction sémantique les deux appels à X_i par un appel à X_{i1} et un autre à X_{i2}.

Enfin, il suffit de générer le système d'équations normalement à partir de cette nouvelle fonction.

Exemple : L'instruction *double* L'instruction **double** exécute deux fois l'instruction qui lui est passée en paramètre.

On associe à la production

$$stat \longrightarrow \textbf{double}\ stlist$$

la fonction sémantique sémantique suivante :

$$\begin{aligned} [\![stat]\!] = \lambda env \cdot \lambda s \cdot \\ \textbf{let}\ s' = [\![stlist]\!](env, s)\ \textbf{in} \\ [\![stlist]\!](env, s') \end{aligned}$$

Cette règle évalue deux fois *stlist*. La première fois, l'état utilisé est celui qui provient de l'exécution de l'instruction précédente, et la deuxième fois, l'état utilisé provient de l'exécution du premier *stlist*.

Le schéma de production généré va être alors :

$$\begin{aligned} &\textbf{with scheme} \\ &\quad stat \longrightarrow st_1 = stlist\ st_2 = stlist \end{aligned}$$

L'expression va être transformée en :

$$\begin{aligned} [\![stat]\!] = \lambda env \cdot \lambda s \cdot \\ \textbf{let}\ s' = [\![st_1]\!](env, s)\ \textbf{in} \\ [\![st_2]\!](env, s') \end{aligned}$$

etla génération des équations va ensuite avoir lieu normalement.

□

2.6.4 Le report des conditions dans les productions conditionnelles

Ce report se fait dans le cas des appels récursifs et de l'évaluation paresseuse. Le problème est le suivant : l'évaluation des conditions les plus intérieures peuvent dépendre des conditions précédentes. C'est pourquoi il faut générer un nombre de productions virtuelles égal, dans le pire des cas, à 2^n où n est le nombre de conditions apparaissant dans l'expression.

Une expression de la forme :

$$\begin{array}{l} \textbf{if}\ cond_1\ \textbf{then} \\ \quad \textbf{if}\ cond_2\ \textbf{then} \\ \qquad \ldots \\ \qquad\quad \textbf{if}\ cond_n\ \textbf{then} \\ \qquad\qquad e_n \\ \qquad\quad \textbf{else} \\ \qquad \ldots \\ \quad \textbf{else}\ e_2 \\ \textbf{else}\ e_1 \end{array}$$

doit être transformée en un ensemble de productions conditionnelles de la forme :

$$
\begin{array}{l}
\textbf{where}\ [\ cond_1 \wedge cond_2 \wedge \ldots \wedge cond_n\ ]\ \ldots \\
\qquad o := e_n \\
\\
\textbf{where}\ [\ cond_1 \wedge \ldots \wedge cond_{n-1}\ ]\ \ldots \\
\qquad o := e_{n-1} \\
\qquad\qquad \vdots \\
\qquad\qquad \vdots \\
\textbf{where}\ [\ cond_1\ ]\ \ldots \\
\qquad o := e_2 \\
\\
\textbf{where}\ [\,]\ \ldots \\
\qquad o := e_1
\end{array}
$$

Pour chacune des expressions e_i, il faut étudier ses dépendances par rapport aux non-terminaux et générer une production virtuelle correspondante. Si plusieurs expressions ont les même dépendances, il est clair que une seule production virtuelle sera générée pour toutes et la distinction sera faite grâce à des productions conditionnelles.

Si l'expression contient un emboîtement plus complexe avec deux branches ou plus, tel que :

$$
\begin{array}{l}
\textbf{if}\ cond_1\ \textbf{then} \\
\qquad \textbf{if}\ cond_2\ \textbf{then} \\
\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad \textbf{if}\ cond_n\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad e_n \\
\qquad\qquad\qquad\qquad \textbf{else if}\ cond\ \textbf{then}\ \cdots \\
\qquad \textbf{else}\ e_2 \\
\textbf{else}\ e_1
\end{array}
$$

on utilise l'opérateur $\neg$ devant la condition du **if** correspondant au **else** posant le problème puis on continue normalement sur l'autre branche.

Exemple : les emboîtements complexes de conditions Cet exemple illustre la méthode décrite ci-dessus pour des emboîtements un peu complexes d'expressions conditionnelles. L'expression ci-dessous va générerles trois productions conditionnelles qui se trouvent plus bas.

if $cond_1$ **then**
  if $cond_2$ **then**
    e_1
  else if $cond_3$ **then**
    e_2
else
  e_3

where $[cond_1 \wedge cond_2]$
  e_1
where $[cond_1 \wedge \neg cond_2 \wedge cond_3]$
  e_2
where $[]$
  e_3

□

Algorithme de génération des productions conditionnelles

Il s'agit de l'algorithme 2.5.

Cette fonction prend en paramètre l'expression à traiter. Elle a pour type

$$\text{condition list} \rightarrow \text{condition list}$$

Fonctions annexes : La fonction **hd**: α list→ α renvoie le premier élément de la liste passée en paramètre.

La fonction **@** : α list→ α list→ α list renvoie la concaténation des deux listes passées en paramètres.

```
let make_cprod expr =
    let rec make_cprod_rec clist expr = match expr with
        IfThenElse(c, t, e) → let lastcond = (hd clist) in
                clist @(make_cprod_rec ([lastcond ∧ c] :: clist) t)
                    @(make_cprod_rec ([lastcond ∧ ¬c] :: clist) e)
    |   Let(id, e1, e2)      →   make_cprod_rec clist e2
    |   _                    →   clist
in make_cprod_rec [true] expr
```

Algorithme 2.5: Génération des productions conditionnelles

Cet algorithme génère une liste de conditions, de la plus générale à la plus particulière.

Le **let** doit être traité comme un cas particulier, car il ne fait qu'introduire de nouvelles variables locales dans l'expression principale, c'est pourquoi il faut le traiter de manière transparente, comme cela est fait dans l'algorithme.

2.7 La transformation d'une sémantique dénotationnelle en grammaire attribuée

Après avoir étudié les problèmes et leur solution dans les sections précédentes, nous nous intéressons à la combinaison des algorithmes pour fournir une méthode globale effectuant la transformation d'une sémantique dénotationnelle en grammaire attribuée. La première étape sera la transformation des domaines en types de données informatiques, suivie par la génération des équations à partir de la fonction sémantique, sous forme d'un algorithme complet, mais qui reprend la méthode de Ganzinger, et les méthodes à appliquer dans le cas où plusieurs des transformations présentées dans la section précédente sont nécessaires. Nous terminons par le cas particulier des valeurs fonctionnelles.

Chaque fonction sémantique est transformée indépendemment des autres, et va nous permettre d'obtenir l'ensemble de règles sémantiques associé à une production concrète.

2.7.1 La transformation des domaines en types de données informatiques

Les domaines sont isomorphes à des types mis à part la notion d'ordre et les éléments $\top$ et $\perp$:

- Les domaines de base, comme D_{bool} et D_{int} deviennent des types de base ;
- les produits cartésiens de domaines sont transformés en tuples dans le cas où il n'y a pas de sélecteur ou bien en enregistrements dans le cas contraire ;
- les sommes de domaines sont transformées en types somme ou union ;
- les domaines fonctionnels deviennent des types fonctionnels.

La notion d'ordre ne nous est pas utile ici et nous nous en passerons mais les éléments $\top$ et $\perp$ représentent des états d'erreur et les types informatiques classiques ne les prennent pas en compte. Il va donc falloir réserver pour chaque type des valeurs particulières pour représenter ces deux éléments. Une autre solution au problème est l'utilisation des exceptions.

2.7.2 L'algorithme de génération des équations

L'algorithme que je décris ici est celui de Ganzinger, mais j'en donne une description détaillée et fonctionnelle.

La fonction *genere_eqs* prend en paramètre un environnement qui associe un non-terminal à une liste de ses paramètres formels, et une expression à partir de laquelle on va générer les équations. Elle est donc du type $(string * string\ list)\ list \rightarrow expression \rightarrow (equation\ list) * expression$.

Cet algorithme est assez évident, si ce n'est pour le cas des fonctions sémantiques, qui nécessite une petite explication : la variable *formal_params* contient la liste des paramètres formels de la fonction que l'on appelle. La variable *effective_params* contient les paramètres effectifs, c'est-à-dire des expressions. Il faut donc générer les équations correspondant à ces expressions, ainsi que leur version modifiée. Il faut ensuite générer les équations correspondant à l'appel que l'on est en train de parcourir, et l'expression de sortie.

Nous disposons maintenant de tous les algorithmes de base nécessaires à cette transformation.

```
let genere_eqs env exp =
    let rec genere_eq_rec = function
        (Id _ | Int _) as x          →  [], x
    |   NonTerm(id, params)  →  let formal_params = assoc id env
                                and effective_params = map
                                        (function x → snd(genere_eq_rec x))
                                        params
                                in
                                    flatten(map
                                        (function x → fst(genere_eq_rec x))
                                        params)
                                    @(map2
                                        (fun x y →
                                            {partie_gauche = x ; partie_droite = y})
                                        formal_params
                                        effective_params),
                                    NonTerm(id,
                                        map (function x → Id x) formal_params)
    |   IfThenElse(c, t, e)      →  let eq_c,exp_c = genere_eq_rec c
                                and eq_t,exp_t = genere_eq_rec t
                                and eq_e,exp_e = genere_eq_rec e in
                                    eq_c@eq_t@eq_e,
                                    IfThenElse(exp_c, exp_t, exp_e)
    in let (eqs, o) = genere_eq_rec exp in
        {partie_gauche ="o"; partie_droite = o}@eqs
```

Algorithme 2.6: Génération du système d'équations

Il faut appliquer cet algorithme pour chacune des fonctions sémantiques, indépendemment les une des autres.

Une première étape consiste à détecter tous les cas posant des problèmes, c'est-à-dire les cas étudiés dans la section précédente. Il faut alors modifier le cas échéant l'expression et générer le système d'équations. Celui-ci va nous permettre de calculer les dépendances, de générer les schémas de productions et les productions conditionnelles, et enfin de générer les règles sémantiques.

2.7.3 Combinaison de plusieurs transformations

Il est tout à fait possible que plusieurs des transformations décrites précédemment soient requises. Je vais donc décrire ici les cas susceptibles de se produire.

Appels récursifs + évaluation paresseuse

Ces deux cas sont en fait très proches l'un de l'autre et peuvent être traités de manière quasiment identique. Leur combinaison ne pose aucun problème. Il suffit en effet d'ajouter à la liste des non-terminaux de la production concernée le non-terminal de la partie gauche.

Appels récursifs/évaluation paresseuse + double appel

Ce cas est plus difficile à traiter. En effet, le traitement du cas des appels récursifs (ou évaluation paresseuse) génère un certain nombre de schémas de production et il faut ensuite tester pour l'expression associée à chacun des schémas si le problème du double appel se pose.

Il est nécessaire de traiter chacun des schémas de cette manière car un double appel se trouvant dans l'expression initiale peut avoir disparu après son "éclatement" dans les différents schémas de production.

Exemple : transformation d'une expression simple Soit l'expression :

$$\begin{array}{l} \textbf{if}\ c_1\ \textbf{then} \\ \qquad [\![X_1]\!](a) \\ \textbf{else} \\ \qquad [\![X_1]\!](b) \end{array}$$

Les deux productions conditionnelles générées à partir de cette expressions vont être :

$$\begin{array}{l} \textbf{where}\ [c_1]\ X_0 \to X_1\ \textbf{use} \\ \qquad o := [\![X_1]\!](a) \\ \textbf{end where;} \\ \\ \textbf{where}\ []\ X_0 \to X_1\ \textbf{use} \\ \qquad o := [\![X_1]\!](b) \\ \textbf{end where;} \end{array}$$

Dans ces deux nouvelles expressions, le problème du double appel n'apparaît plus et il n'est donc plus nécessaire de le traiter.

Si le problème du double appel était apparu, il aurait fallu appliquer la transformation classique à chacune des productions conditionnelles, *indépendemment l'une de l'autre.* □

2.7.4 Simplification et génération de la grammaire attribuée finale

La simplification du système d'équations a lieu comme indiqué dans la section 2.5.2. D'autres simplifications peuvent être effectuées en réduisant les λ-expressions et en appliquant des règles de simplification classiques. Je ne les détaillerai pas ici car ce n'est pas l'objet principal de mon stage.

La génération de la grammaire attribuée a lieu quant à elle comme indiqué dans l'article de Ganzinger. Il faut associer chacune des variables du système d'équations à un non-terminal, et on obtient la grammaire attribuée.

2.7.5 Traitement des valeurs fonctionnelles

Dans l'état actuel des choses, le langage Olga ne permet pas traiter les fonctions d'ordre supérieur. Voici cependant la méthode qu'il faudrait utiliser pour résoudre le problème.

La méthode permettant de traiter ce problème est assez simple : une valeur fonctionnelle correspond en fait à un sous-arbre dont on ne veut pas évaluer les attributs immédiatement, souvent parce que l'un des attributs hérités de ce sous-arbre est inconnu et ne le sera que plus tard. Il faut donc stocker ce sous-arbre et appeler l'évaluateur de grammaires attribuées dessus lorsque la valeur de l'attribut sera connue.

On peut considérer que cela revient à déplacer le sous-arbre de l'endroit où il apparaît syntaxiquement pour le mettre à l'endroit où l'on veut l'évaluer. Cet endroit n'est pas connu à la compilation mais uniquement à l'évaluation car lors de la compilation, l'arbre abstrait du programme à interpréter n'est pas disponible. Cela peut fonctionner dans les cas simples, comme le **goto** mais lors de l'appel de procédures, cela devient très compliqué.

Dans le cas de la sémantique dénotationnelle, les valeurs fonctionnelles sont surtout utilisées pour les continuations et sont donc indispensables lorsque l'on veut traiter les branchements inconditionnels (**goto**s) et les appels de procédures.

Chapitre 3

L'implémentation du traducteur

Ce chapitre présente l'implémentation des méthodes décrites dans le chapitre précédent. Je commence par décrire le système FNC-2 , dans lequel j'ai réalisé l'implémentation et je continue par la description du langage source utilisé pour spécifier la sémantique dénotationnelle, puis la manière dont j'ai réalisé le traducteur dans le langage Olga.

3.1 Le système FNC-2

FNC-2 est un système permettant l'écriture et l'exécution de grammaires attribuées. Les grammaires attribuées sont écrites dans un langage spécialisé, Olga. Les arbres d'entrée et de sortie sont décrits dans le langage ASX et compilés. La syntaxe concrète est décrite à l'aide du système SYNTAX, qui peut être comparé à Lex et Yacc mais qui est bien plus puissant. Le résultat de l'évaluation d'une grammaire attribuée peut être visualisé grâce à Ppat, qui génère une forme textuelle à partir d'un arbre abstrait. L'interfaçage entre FNC-2 et Syntax se fait par l'intermédiaire du processeur ATC qui génère un arbre abstrait.

Le système FNC-2 permet d'enchaîner l'exécution de plusieurs grammaires attribuées sur un arbre abstrait qui sera transformé ou décoré au fur et à mesure.

On peut voir une grammaire attribuée comme une fonction qui prend en entrée un arbre de syntaxe abstraite décoré par des attributs, et qui rend un (ou plusieurs) arbres abstraits avec d'autres attributs. Les deux arbres peuvent avoir la même structure avec uniquement les attributs qui diffèrent, ou bien avoir une structure totalement différente ; par exemple une grammaire attribuée pourrait prendre en entrée l'arbre d'un programme source quelconque et générer un arbre correspondant au code intermédiaire, dans un compilateur.

Le compilateur FNC-2 peut générer du code C, ML ou Lisp.

3.1.1 Le système Syntax

La description de la syntaxe d'un langage se fait par l'intermédiaire de quatre spécifications :

LECL : il s'agit de l'analyseur lexical, qui permet de décrire les différents lexèmes à l'aide d'expressions régulières.

RECOR : il s'agit du système de gestion des erreurs et permet de décrire la manière dont les erreurs d'analyse syntaxique doivent être traitées.

PRIO : il s'agit d'un système permettant de spécifier la priorité des opérateurs dans le langage source. Cela permet de ne pas avoir à l'indiquer à l'aide de constructions BNF dans la grammaire.

3.1.2 Le langage Olga

C'est dans ce langage que l'on spécifie les grammaires attribuées. Il s'agit d'un langage fonctionnel du premier ordre, modulaire et fortement typé.

Les grammaires attribuées peuvent être considérées comme des fonctions qui prennent en paramètre un arbre de syntaxe abstraite, et qui renvoient zéro, un ou plusieurs arbres de syntaxe abstraite. Ces arbres sont décrits par des fichiers ASX. Il est possible, grâce à la meta-composition, d'effectuer une composition fonctionnelle de ces grammaires attribuées qui élimine les arbres intermédiaires.La meta-composition est étudiée dans [Rou94].

Les types de données habituels sont implémentés (entiers, booléens...) ainsi que les types construits pour les listes, les enregistrements et les unions.

Olga permet de plus de manipuler les syntaxes abstraites décrites dans les fichiers ASX, et fournit les constructeurs pour les arbres.

Les règles sémantiques définissant les attributs peuvent être générées automatiquement dans certains cas (les règles de copies en particulier), ce qui rend les grammaires attribuées plus claires et plus simples à écrire.

Une description détaillée du système FNC-2 se trouve dans [Viv96] et son manuel se trouve dans [JP].

3.2 L'implémentation du traducteur

Ce traducteur doit pour le moment être considéré comme un prototype. Il n'implémente en effet qu'un sous-ensemble très restreint du meta-langage utilisé pour décrire une sémantique dénotationnelle. Les domaines disponibles se limitent aux types de base et aux tuples. Les expressions se limitent à quelques constructions de bases : la conditionnelle, le **let** et l'application aux fonctions normales et aux fonctions sémantiques.

Le traducteur de sémantique dénotationnelle en grammaire attribuée s'intègre dans le système FNC-2 comme cela est indiqué dans la figure 3.1. Les boites en grisé représentent les fichiers que doit fournir l'utilisateur. Le traducteur génère un fichier olga et un nouveau fichier asx qui seront alors compilés normalement par FNC-2 .

Le fichier d'entrée ASX décrit la syntaxe abstraite du langage source et le fichier DENOT décrit la sémantique dénotationnelle.

3.2.1 Le langage source

Le langage source est basé sur les expressions purement fonctionnelles du premier ordre pour les fonctions sémantiques.

Dans l'état actuel des choses, aucune gestion d'erreur n'est effectuée.

Un source se divise en trois parties :

- les déclarations et les importations ;
- les déclarations des domaines des non-terminaux ;

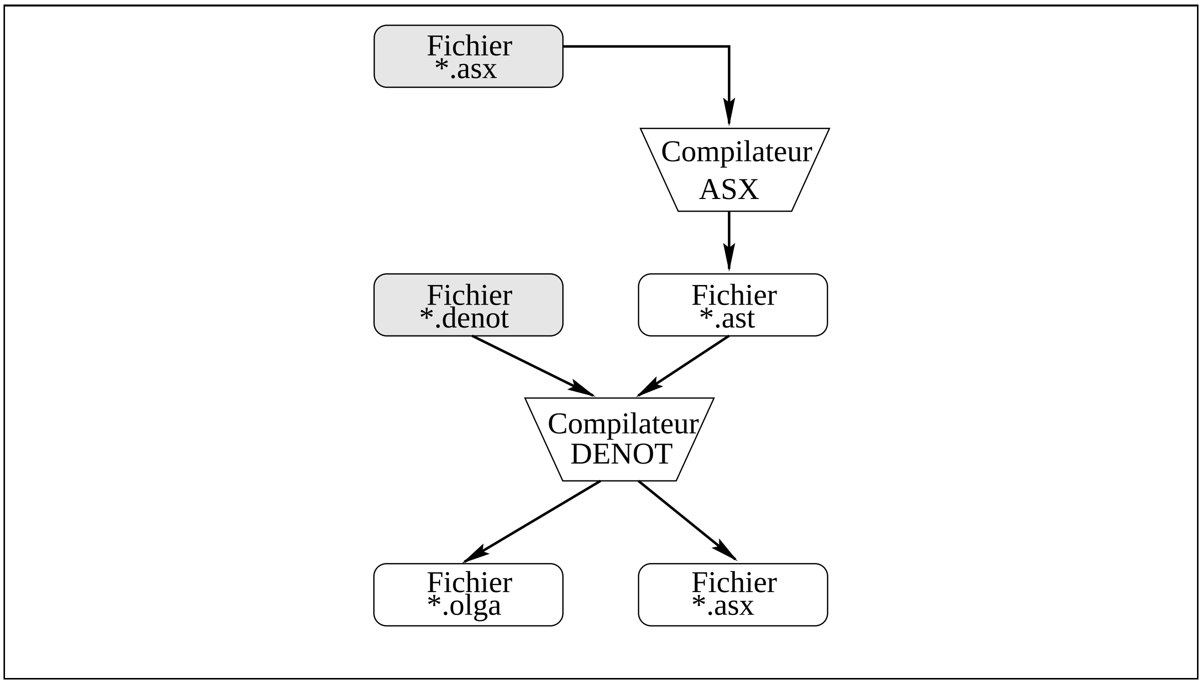

FIG. **3.1** – Intégration du traducteur dans le système FNC-2

– les règles associées à chaque production.

Les déclarations et les importations

Cette partie comprend l'en-tête du fichier et les importations de modules Olga. L'en-tête doit inclure le nom de la sémantique et le nom d'une grammaire abstraite, sur laquelle est basée la sémantique dénotationnelle.

Les importations de modules permettent d'importer des modules Olga complets ou uniquement certaines fonctions. Les types des fonctions importées doivent être définissables dans la sémantique dénotationnelle, c'est-à-dire les entiers, les booléens et les tuples.

Les déclarations des domaines des non-terminaux

Chaque phylum se voit associé un domaine fonctionnel, qui est donc un couple de types. Il est nécessaire de donner un type d'entrée et un type de sortie. Les types possibles sont les entiers, les booléens et les tuples.

Les opérateurs sur les domaines seront ici remplacées par les notations équivalentes en programmations fonctionnelle :

– *var* **in** D est remplacé par les constructeurs de types.

– *var* **to** D est remplacé par le déréférencement des constructeurs de types.

Ces notations sont celles de l'article de Ganzinger.

Mais dans l'état actuel des choses, ces deux opérations ne sont pas implémentées.

De plus, les éléments $\top$ et $\perp$ ne sont pas gérés.

Les règles associées à chaque production

Ces règles reprennent la production et une valeur fonctionnelle, qui doit être du type déclaré pour le phylum associé à l'opérateur de la partie gauche de la production.

3.2.2 Le compilateur

Le compilateur fonctionne en quatre passes :

- la vérification des types et de la cohérence par rapport à la grammaire abstraite,
- la détection des cas difficiles et la transformation des expressions,
- la génération des systèmes d'équations et l'optimisation des expressions, et
- la génération de la grammaire attribuée, c'est-à-dire du fichier Olga et du fichier ASX.

Chacune de ces passes est implémentée par une grammaire attribuée. Leur exécution s'enchaîne et l'arbre d'entrée de chaque grammaire attribuée correspond à l'arbre de sortie de la grammaire attribuée précédente. L'arbre d'entrée de la première grammaire attribuée provient du code source de la sémantique dénotationnelle.

Vérification des types

Cette grammaire attribuée vérifie la cohérence des types entre la déclaration des domaines des phyla et les fonctions sémantiques elles-mêmes. Ne serait-ce que pour cette raison, elle a besoin d'aller regarder dans un fichier ASX pour savoir à quel phylum correspond un opérateur donné.

De plus, elle vérifie la cohérence entre les productions de la sémantique dénotationnelle et celles de la grammaire abstraite décrite dans le fichier ASX.

Le fonctionnement général consiste à stocker les types déclarés pour chaque phylum dans une table des symboles puis à comparer ce type à celui que l'on synthétise pour chacune des fonctions sémantiques.

La détection des cas difficiles

Cette grammaire attribuée recherche les cas difficiles pour chacune des fonctions sémantiques du programme, et génère des schémas de production et les productions conditionnelles.

Une première étape consiste à détecter le cas de l'évaluation paresseuse, car le traitement des autres cas dépend de celui-ci. Différentes dépendances sont aussi calculées, et en particulier, les dépendances des expressions et des sous-expressions par rapport aux non-terminaux. En effet, lorsque l'évaluation paresseuse est nécessaire, l'expression se retrouve éclatée dans les différents schémas de production et il faut donc dans chacun de ces schémas de production déterminer quels sont les non-terminaux utilisés. Ceux-ci fournissent en plus les informations dont nous avons besoin pour le traitement du cas du double-appel et du cas des appels récursifs.

La deuxième étape consiste à générer les schémas de production eux-même ainsi que les productions conditionnelles. On le fait simultanément pour les trois cas particuliers, en parcourant l'arbre syntaxique de l'expression et en générant les schémas au fur et à mesure.

L'arbre de syntaxe abstraite qui est généré contient donc toutes les informations nécessaires à la génération du système d'équations et de la grammaire attribuée : les dépendances, les schémas de production et les productions conditionnelles.

La génération des systèmes d'équations

Cette grammaire attribuée doit extraire si cela est nécessaire les sous-expressions à transformer pour chacun des schémas de production de l'expression principale de la fonction sémantique. Chacune de ces sous-expressions sera ensuite transformée en un système d'équations.

Les optimisations

C'est ici que l'on traitera le cas de la factorisation et que l'on simplifiera les systèmes d'équations.

A la fin de cette étape, il n'y aura plus qu'à générer la grammaire attribuée.

La génération des fichiers de sortie

Nous avons donc toutes les informations nécessaires à la génération des systèmes d'équations. Une fois ces systèmes d'équations générés, nous avons toute l'information nécessaire pour calculer les dépendances, connaître les attributs, les non-terminaux de toutes les productions et nous sommes donc prêts à générer le fichier Olga et le fichier ASX. Nous utiliserons le processeur *PPAT*, dont le but est de décompiler un arbre de syntaxe abstraite.

Conclusion

L'article de Ganzinger nous a donné une méthode permettant de convertir une spécification écrite en sémantique dénotationnelle en une grammaire attribuée. Cependant, les restrictions qu'il a imposé à la grammaire attribuée générée rendaient le processus difficilement utilisable pour une implémentation de sa méthode. Depuis quinze ans, personne n'a travaillé sur l'approche de Ganzinger.

Les grammaires attribuées dynamiques développées récemment ont permis de répondre parfaitement à ces problèmes en proposant des extensions permettant de simuler l'évaluation paresseuse et les schémas récursifs. Nous avons montré dans nos travaux comment utiliser ces extensions pour étendre la méthode de Ganzinger.

Il reste néanmoins beaucoup de choses à faire, en particulier prouver la correction de la transformation, c'est-à-dire prouver que la grammaire attribuée générée avec notre méthode est sémantiquement équivalente à la sémantique dénotationnelle de départ.

Il faut aussi soulever le problème de l'ordre supérieur, utilisé fréquemment en sémantique dénotationnelle, dont la traduction en grammaire attribuée n'est pas bien définie. Les grammaires attribuées d'ordre supérieur doivent permettre les appels récursifs de l'évaluateur sur un sous-arbre. L'ordre supérieur dont nous avons besoin pour effectuer la traduction est assez simple, mais le cas général est beaucoup plus complexe.

Nous avons présenté dans ce rapport une méthode permettant de rendre exécutable une spécification en sémantique dénotationnelle. Les avantages que cela procure sont nombreux :

- l'écriture d'un compilateur se résume à spécifier la syntaxe concrète du langage source, et sa sémantique dénotationnelle ;
- la spécification est parfaitement respectée et son formalisme permet d'éviter les ambiguités ;
- il est beaucoup plus simple de faire des preuves sur une sémantique dénotationnelle que sur d'autres formalismes, et il est donc possible de prouver des propriétés du compilateur qui seront vérifiées par l'exécutable ;
- de nombreuses erreurs dans l'écriture des compilateurs sont éliminées grâce à l'automatisation du processus.

Le fait de transformer une sémantique dénotationnelle en grammaire attribuée permet de profiter de la puissance des analyses statiques qu'il est possible d'appliquer aux grammaires attribuées et donc d'obtenir des interpréteurs bien optimisés.

Au delà de la sémantique dénotationnelle, ce travail est aussi une première étape vers la traduction des langages fonctionnels en grammaires attribuées. En effet, les meta-langages utilisés pour la sémantique dénotationnelle sont des langages fonctionnels.

Cela peut se révéler particulièrement intéressant car la meta-composition des grammaires attribuées, qui permet d'effectuer la composition fonctionnelle de deux grammaires attribuées, considérées comme des fonctions prenant en entrée un arbre syntaxique et rendant un nouvel arbre syntaxique, permet d'éliminer l'arbre syntaxique intermédiaire (voir [Rou94]). Si on applique la meta-composition à des grammaires attribuées provenant de programmes fonctionnels, on pourrait obtenir une méthode permettant d'éliminer les structures intermédiaires dans un programme fonctionnel. Cette idée est développée dans [DPRJ96]. Cet article est une comparaison entre les grammaires attribuées et la théorie des folds, c'est-à-dire les itérateurs sur des types construits en programmation fonctionnelle.

Bibliographie

[Boy96] John Boyland. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, January 1996.

[DPRJ96] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2957.ps.gz`.

[Gan80] Harald Ganzinger. Transforming denotational semantics into practical attribute grammars. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, New York–Heidelberg–Berlin, 1980.

[JP] Martin Jourdan and Didier Parigot. *The* FNC-2 *System User's Guide and Reference Manual.* INRIA, Rocquencourt, 1.9 edition.

[KLMM83] Gilles Kahn, Bernard Lang, Bertrand Mélèse, and Éliane Marcos. Metal: a formalism to specify formalisms. *—SCICP—*, 3:151–188, 1983.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: Mathematical Systems Theory 5, 1, pp. 95-96 (March 1971).

[Mad80] Ole L. Madsen. On defining semantics by means of extended attribute grammars. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 259–299. Springer-Verlag, New York–Heidelberg–Berlin, 1980. See also: report DAIMI PB-109, Comp. Sc. Department, Aarhus University (January 1980).

[May81] Brian H. Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3):503–518, August 1981. See also: report DAIMI PB-90, Comp. Sc. Department, Aarhus University (August 1978).

[Paa95] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[PDRJ95] Didier Parigot, Étienne Duris, Gilles Roussel, and Martin Jourdan. Attribute grammars: a declarative functional language. rapport de recherche 2662, INRIA, October 1995. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2662.ps.gz`.

[PRJD96] Didier Parigot, Gilles Roussel, Martin Jourdan, and Étienne Duris. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2881.ps.gz`.

[Rou94] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[Viv96] Bruno Vivien. Etude du système syntax/FNC-2 pour la génération de compilateurs. Technical report, Conservatoire National des Arts et Métiers, June 1996.