Sémantique Équationnelle

Loïc Correnson

28 février 2000

Table des matières

Chapitre 1

Introduction

Cette thèse s'inscrit dans le cadre des méthodes permettant la transformation de programmes. Le but des ces opérations est d'optimiser le fonctionnement du programme transformé. C'est un sujet assez vaste car il existe de nombreuses façons de procéder qui peuvent grandement varier d'un style de programmation à un autre. Aujourd'hui, la plupart des compilateurs réalisent de nombreuses optimisations que l'on peut considérer comme des transformations de programmes. Je ne traiterai pas de ce type d'optimisations dans la mesure où elles sont réalisées sur des programmes écris avec des langages d'un niveau proche du code machine ou du code intermédiaire. J' m'interesse ici à une autre gamme de transformations qui visent à optimiser un programme écris dans un langage de haut niveau.

Ce type de transformations dans les langages de haut niveau est particulièrement important, par exemple quand on a programmé des algorithmes complexes de manière modulaire et que l'on voudrait en obtenir une version simplifiée. Ainsi, une fonction f aura été programmé comme la composition des fonctions f_1 et f_2, ou bien comme un cas particulier d'une fonction g dont certains paramètres seraient fixés. Dans les cas simples, on pourra optimiser à la main les programmes ainsi construits. Mais dans les cas plus complexes il est difficile de trouver la simplification et il faut recourir à des méthodes automatiques.

Comme je l'ai déjà mentionné, les méthodes de transformations dépendent fortement du type de programmation et du langage considéré. Cependant, on trouve souvent un air de famille entre diverses méthodes. Cette thèse a pour objectif d'unifier les résultats obtenus par des techniques de transformations provenant de deux paradigmes différents: la programmation fonctionnelle et les grammaires attribuées.

1.1 Programmation Fonctionnelle

Je suppose que les notations fonctionnelles sont déjà familière au lecteur. Cependant, comme c'est un domaine qui contient des langages assez variés, j'aurai besoin de fixer un langage fonctionnel de référence pour pouvoir faire des développements formels et énoncer des théorèmes. Ce langage de référence sera d'ordre supérieur mais sans effets de bords, ne sera pas fortement typé, et sera muni d'une sémantique d'appel par valeur. Je consacrerai un chapitre à rappeler ces définitions.

Les transformations fonctionnelles qui m'intéressent sont celles qui ont trait à la composition de fonctions et à l'évaluation partielle. Pour illustrer ce type de transformations, observons un exemple sur des listes. Une liste est soit la liste vide (notée `nil`), soit la liste constituée d'un élément `e` et d'une liste `l` (notée `cons e l`). La très classique fonction `map` permet d'appliquer une fonction `f` à tous les éléments d'une liste. Elle est définie ainsi:

```
let map f x = match x with
  | cons e l -> cons (f e) (map f l)
  | nil -> nil
```

Cette fonction est très souvent utilisée comme brique de base permettant de construire des fonctions plus complexes. Considérons par exemple la fonction suivante:

```
let complex f x = map (fun x -> 1+x) (map f x)
```

Cette fonction fait intervenir une composition des deux fonctions `map`. Son calcul est laborieux: il nécessite tout d'abord la construction d'une liste intermédiaire créée par `(map f x)`. Cette liste intermédiaire est alors consommée par `(map (fun x ->1+x))`. Une simplification peut être obtenue et l'on peut constater que la fonction `complex` peut être aussi écrite de la manière suivante:

```
let complex f x = match x with
  | cons e l -> cons (1+(f e)) (complex f l)
  | nil -> nil
```

Cette version est bien plus efficace pour plusieurs raisons: on économise la construction de la liste intermédiaire, ainsi que de nombreux appels de fonctions qui sont liés au fait que le calcul de `(1+(f e))` est, dans la version initiale, décomposé en deux fonctions distinctes.

L'opération qui consiste à passer de la première version de la fonction `complex` à la seconde est appelée d'une manière générale une “déforestation”. Cette appellation provient du fait que l'on supprime par ces transformations des calculs intermédiaires qui sont en général des arbres plus ou moins compliqués (les listes sont un cas particulier d'arbres).

Malheureusement, cette opération n'est pas facile à automatiser. Il existe des cas que l'on ne sait pas résoudre autrement qu'avec une intervention manuelle.

On peut alors distinguer différentes catégories de méthodes de déforestation sur lesquelles je reviendrai au cours de ma thèse:

- Les méthodes purement fonctionnelles, qui modifient astucieusement le programme initial de manière à faire apparaître des simplifications. Chronologiquement, l'une des premières approches en a été donnée par l'algorithme de Wadler [43, 44]. Ces méthodes, largement explorées [38, 37, 2, 34], ne marchent que dans certains cas. Toute la difficulté réside alors dans la détection des cas favorables, ou du moins ceux qui vont aboutir à une solution.
- Les méthodes sémantiques qui transforment tout d'abord le programme fonctionnel initial dans un autre formalisme (équivalent), mais dans lequel les compositions sont plus facilement simplifiées. Ces méthodes sont souvent construites avec des notions catégorielles, notamment des folds et plus généralement des hylomorphismes [22, 10, 36, 40, 19, 13, 12, 11, 35, 24].

Quelle que soit la méthode utilisée, les cas traités sont similaires et les situations qui s'avèrent difficile pour une méthode le sont aussi pour les autres méthodes. Par exemple, la composition de l'inversion de liste avec elle-même est un cas d'école réputé difficile. Prenons une définition avec accumulateur de la fonction d'inversion:

```
let rev h x = match x with
  | cons a b -> rev (cons a h) b
  | nil -> h
let reverse x = rev nil x
```

Ici, la fonction auxiliaire `rev` réalise l'inversion en empilant les éléments de la liste sur son paramètre `h` qui joue alors le rôle d'accumulateur. La composition s'écrit simplement:

```
let compo x = reverse (reverse x)
```

La simplification attendue est bien entendu de supprimer tout calcul car la fonction `compo` est tout simplement égale à l'identité. Cet exemple, pourtant simple, n'est pas résolu automatiquement par les méthodes fonctionnelles. Pour les algorithmes utilisant l'évaluation partielle, aucune modification n'est trouvée. Pour les méthodes à base de folds et d'hylomorphismes, il existe des solutions mais le programmeur doit explicitement donner des informations supplémentaires aux algorithmes de transformations (voir par exemple [36]).

Je reviendrais plus tard sur ces cas difficiles que l'on peut caractériser par la présence d'accumulateurs, une situation hélas fréquente en programmation fonctionnelle.

1.2 Grammaires Attribuées

Ce formalisme a été initialement introduit pour faciliter l'écriture des compilateurs. C'est un style de programmation très performant pour spécifier des algorithmes travaillant sur des arbres définis par une grammaire, domaine qui ne se limite donc pas aux compilateurs. C'est un formalisme solide bénéficiant d'un large contexte d'études, d'analyses, d'implémentations et de mise en oeuvre [25, 16]. Intuitivement, pour chaque règle de la grammaire on définit sous forme d'équations orientées les calculs à effectuer. Ainsi, les listes sont des arbres représentés par la grammaire suivante (avec ici une syntaxe très informelle):

```
List ::= cons Element List
       |  nil
```

Sur cette grammaire, on peut définir la longueur d'une liste en écrivant la grammaire attribuée suivante:

```
cons e l -> length = 1 + b.length
nil -> length = 0
```

Ici, `length` est un attribut que l'on calculera pour toute liste, et étant donnée une liste `l` on représentera par `l.length` la valeur de l'attribut `length` qui lui est associé. La signification des deux équations précédentes est que l'attribut `length` de toute liste vide est égal à 0, alors que si la liste est sous la forme `cons e l`, sont attribut `length` est égal à 1 plus l'attribut `length` de la liste `l`. Le travail d'un compilateur de grammaire attribuée est de trouver à partir de ces équations un algorithme permettant de calculer la valeur des attributs en chaque point d'une liste. De nombreux travaux ont été menés pour construire ces évaluateurs [4, 26, 16, 27]. Dans le cas où cet algorithme serait écrit en langage fonctionnel, on aurait par exemple pour l'attribut `length` la fonction suivante:

```
length x = match x with
  | cons e l -> 1 + (length l)
  | nil -> 0
```

Comme on peut le constater sur cet exemple, il y a un certain air de famille entre programmes fonctionnels et grammaires attribuées. Du côté des méthodes de transformation de programme, il y a pourtant un fossé très large. Les transformations de grammaires attribuées sont moins variées que pour les programmes fonctionnels, mais en revanche, la déforestation des cas d'école réputés difficiles pour la programmation fonctionnelle sont des cas faciles pour les grammaires attribuées. Inversement, les cas faciles en programmation fonctionnelle le sont également en grammaires attribuées. L'algorithme de déforestation pour les grammaire attribuées a été inventé par Ganzinger et Giegerich [7] puis développée par Roussel [31] et les comparaisons avec les méthodes fonctionnelles ont été étudiées par Duris [5].

Cependant, il y a une autre différence de fond entre les deux formalismes qui explique cette différence de performance des méthodes de déforestation. En effet, le formalisme des grammaires attribuées a un pouvoir d'expression beaucoup moins grand, c'est à dire que toute fonction ne peut pas être traduite en une grammaire attribuée, alors que l'inverse est vrai. Les fonctions que l'on ne peut pas traduire sont tout simplement celles dont la complexité n'est pas linéaire en la taille de leur arbre d'entrée. On ne peut donc pas espérer plonger les programmes fonctionnels dans les grammaires attribuées pour les déforester.

1.3 Unification des méthodes

Pour unifier les méthodes de déforestation et leur performances respectives, il faut donc dans un premier temps trouver un formalisme commun aux deux paradigmes que sont la programmation fonctionnelle et les grammaires attribuées. C'est un sujet qui a déjà été exploré, notamment par des extensions du formalisme des grammaires attribuées lui permettant de combler son manque d'expression [41, 42, 39, 1, 28, 29, 18]. Ces méthodes n'ont pas toujours été développées à des fins de déforestation, sauf dans le cas de [18] qui utilise la notion de Tree Transducer [17]. Grâce à ce formalisme, on peut exprimer des algorithmes à la manière des grammaires attribuées mais avec des attributs qui peuvent eux-mêmes être des fonctions. On a alors de nombreux théorèmes de conversion et de normalisation qui permettent de reconnaître des cas de déforestation ne faisant apparaître que des grammaires attribuées classiques, ou des cas de déforestation utilisant des algorithmes d'évaluation partielle de type fonctionnel.

Mais pour ces différentes solutions, je me permets de faire deux remarques qui m'ont incité à poursuivre cet effort de recherche sur une autre voie.

Premièrement, l'intégration avec la programmation fonctionnelle s'arrête souvent au premier ordre, l'ordre supérieur étant sujet à d'autres extensions. Deuxièmement, l'unification des méthodes de déforestation, quand elle est recherchée, s'arrête à l'application des cas relevant de l'un ou l'autre formalisme de départ. Dans ce cadre de recherche, mon travail de thèse peut être découpé en trois parties:

Sémantique

Je définis dans cette thèse un nouveau formalisme, que je nomme *Sémantique Équationnelle* et qui est fortement inspiré des grammaires attribuées, mais qui est à mon avis plus basique. En effet, ce n'est pas une extension mais plutôt un appauvrissement des grammaires attribuées, qui bien entendu les contient et dans lequel les langages fonctionnels d'ordre supérieur peuvent être intégralement plongés. Ce travail comprend la définition du formalisme, de sa sémantique, d'une sémantique opérationnelle, et d'une traduction vers et depuis les langages fonctionnels d'ordre supérieur.

Déforestation

Je présente une nouvelle méthode de déforestation, dans laquelle on peut reconnaître des composants issus de différentes méthodes. Ainsi, la méthode de déforestation des grammaires attribuées a été coupée en deux parties: l'une est parfaitement similaire à l'évaluation partielle sur les programmes fonctionnels classiques; l'autre correspond à la déforestation au travers des accumulateurs. La deuxième partie, cependant, ne s'applique plus uniquement à des grammaire attribuées, mais a été étendue à n'importe quel programme de la sémantique équationnelle.

Mise en oeuvre

Pour illustrer ma démarche et mettre au point ma méthode de déforestation, j'ai implémenté un prototype d'outil permettant de déforester automatiquement des programmes fonctionnels d'ordre supérieur. A ce travail est associé un effort important destiné à réduire la complexité des algorithmes utilisés et rendre la méthode opérationnelle.

Comparativement aux autres approches mentionnées, j'ai essayé de simplifier les moyens mis en oeuvre pour unifier différentes méthodes de transformation de programme. Ainsi, mon formalisme de *sémantique équationnelle* est très simple: sa syntaxe et sa sémantique sont d'une complexité comparable à celle du λ-calcul non typé, formalisme qui sert de support théorique aux langages fonctionnels. Par ailleurs, les différentes méthodes de déforestation provenant d'origines différentes n'ont pas été simplement juxtaposées. A savoir que les méthodes provenant des grammaires attribuées ne se limitent pas au sous-ensemble des programmes de la sémantiques équationnelle qui sont des grammaires attribuées, mais qu'elles s'appliquent l'ensemble des programmes fonctionnels (voir par exemple la section 5.5). Ainsi, les parties communes à différentes méthodes ont été fusionnées (voir section 5.4).

1.4 Tour d'horizon

Avant de plonger au coeur de cette thèse, je voudrais en dresser un sommaire et donner quelques indications de lecture. Dans un premier temps, au chapitre 2, je fixe un langage fonctionnel de référence et je présente les méthodes classiques de transformations fonctionnelles. Dans le chapitre 3, je définis ma sémantique équationnelle, ainsi que les principales notions qui forment la charpente de ce formalisme. J'y présente également un plan détaillé du travail mis en oeuvre pour la construction de ma méthode de déforestation. Enfin, le chapitre 4 en présente quelques résultats.

Les autres chapitres portent chacun sur un aspect particulier de mon travail. Ainsi, le chapitre 5 décrit uniquement les méthodes de déforestation et de transformation de programme. Ces méthodes sont correctes, c'est à dire que les transformations effectuées sur un programme ne changent pas sa sémantique. Les preuves de correction sont construites autour de trois principes ou théorèmes élémentaires que je démontre au chapitre 6. Ces preuves sont complexes, mais une fois acceptées, elles permettent de fabriquer facilement des transformations de programme dont la preuve de correction est facile à obtenir. Le chapitre 7 est consacré à la mise en oeuvre et à l'implémentation d'une transformation particulière qui forme l'un des composants principaux des autres transformations. J'ai appelé abusivement *évaluation partielle* cette transformation parce que c'est une sorte de dénominateur commun à beaucoup de méthodes fonctionnelles utilisant l'idée d'une exécution partielle d'un programme, notamment celle nommée *évaluation partielle* dans la communauté de programmation fonctionnelle.

Pour ce qui est de la mise en oeuvre d'un prototype implémentant ces techniques de déforestation, elle m'a permis de définir un autre point de vue sur la sémantique équationnelle, plus tourné vers la conduite effective d'un calcul et en définitive, vers la traduction en programmes fonctionnels. Je consacre le chapitre 8 à ce sujet. Dans le même ordre d'idée, le chapitre 9 rapporte tout ce qui concerne la traduction des programmes fonctionnels en programmes équationnels.

On trouvera à la fin de la thèse le chapitre 10 qui regroupe une série d'annexes sur les relations, les graphes, quelques algorithmes originaux, etc. Enfin, le chapitre 12 est un index des définitions qui permettra, je l'espère, au lecteur de ne pas être perdu.

Chapitre 2

Programmes Fonctionnels

Dans ce chapitre, je vais rapidement présenter un langage fonctionnel d'ordre supérieur simple qui me servira de référence au long de ma thèse. D'un point de vue technique, sa syntaxe est volontairement très limitée. Cependant il m'arrivera d'utiliser de nombreuses extensions que l'on pourra considérer comme du "sucre" syntaxique. La sémantique sur laquelle s'appuie ce langage est une sémantique d'appel par valeur dans laquelle l'ordre d'évaluation des arguments d'une fonction n'est pas fixé, et il n'y a pas d'effets de bords.

Dans la suite du chapitre, je présenterai succintement les éléments clefs des algorithmes classiques de déforestation et de transformation de programme, ainsi que les cas problématiques.

2.1 Constructeurs et Primitives

Une valeur ou une expression manipulée par un programme fonctionnel est appelée en général un *terme*. Cette section concerne la structure et la formation des termes.

2.1.1 Constructeurs

Si $t_1 \ldots t_n$ sont des termes, on peut construire un nouveau terme à l'aide d'un constructeur f en formant l'expression $(f\ t_1 \ldots t_n)$. Ici, f est simplement un nom. L'entier n peut aussi être nul, auquel cas le terme se réduit au seul nom du constructeur, par exemple (f).

2.1.2 Exemple: les listes

Les listes, comme $[a; b; c; d]$ sont représentées par des termes construits avec les constructeurs `nil` et `cons`. Une liste vide est représentée par le terme (`nil`). Un terme construit avec (`cons` $e\ l$) représente une liste dont le premier élément est e et dont tous les éléments suivants forment une liste représentée par l.

Ainsi, la liste $[a; b; c]$ est représentée par:

$$(\texttt{cons}\ a\ (\texttt{cons}\ b\ (\texttt{cons}\ c\ \texttt{nil})))$$

Ici, j'ai considéré que a, b et c étaient trois constructeurs d'arité nulle.

en effet, soit `zero` le constructeur qui représente l'entier 0, les autres entiers sont de la forme (`succ` n), avec n un autre entier de Péhano et `succ` étant le constructeur du successeur de n. Ainsi, l'entier 2 est représenté par le terme (`succ` (`succ` (`zero`))).

2.1.3 Primitives

Un programme fonctionnel peut effectuer des calculs arithmétiques, il faut donc des termes qui représentent des entiers ou des réels (en virgule flottante). On peut coder les entiers avec des constructeurs: en effet, soit `zero` le constructeur qui représente l'entier 0, les autres entiers sont de la forme (`succ` n) pour le successeur de n. Ainsi, l'entier 2 est représenté par le terme (`succ` (`succ` (`zero`))). Cependant, ce n'est pas une manière très commode ni très réaliste de représenter les entier. On pourrait généraliser avec les réels en virgule flottante ou tout autre opération de base. J'utilise donc une notion (classique) d'opérations primitives.

Les opérations élémentaires comme l'addition, la multiplication (etc.) sont aussi représentées par des primitives. J'utiliserai souvent une notation polonaise préfixe par homogénéité avec la notation des constructeurs. Ainsi $2+4*5$ s'écrira $(+\ 2\ (*\ 4\ 5))$.

Les termes primitifs ne sont pas restreints aux entiers mais peuvent concerner tout autre type de valeurs.

Définition 2.1.1 (Termes)
Soit Π *un ensemble de primitives,* $\mathcal{C}ons$ *un ensemble de constructeurs, je définis l'ensemble* $\mathcal{T}erms$ *des termes par le plus petit ensemble tel que tout élément est soit de la forme* $(\pi\ t_1 \ldots t_n)$*, soit de la forme* $(c\ t_1 \ldots t_n)$*, avec* $(\forall i)\ t \in \mathcal{T}erms$*,* $\pi \in \Pi$*,* $c \in \mathcal{C}ons$ *et* n *pouvant être nul.*

2.2 Programmes Fonctionnels

Un programme fonctionnel est un ensemble de déclarations de fonctions mutuellement récursives. Chaque fonction possède un nom qui la caractérise. Une fonction associe à une ou plusieurs variables une expression. Par exemple, la fonction `inc` qui incrémente sa variable est définie par:

```
let inc x = (+ 1 x)
```

Une expression est simplement un terme qui peut contenir des variables ou des appels de fonctions. Pour incrémenter 4 on écris par exemple (`inc` 4). On peut définir des fonctions de plusieurs variables et faire des applications partielles, ce qui donne donc un langage d'ordre supérieur.

Pour les termes construits avec des constructeurs, on peut définir des fonctions par cas sur le constructeur de tête d'une variable. Alors cette variable n'est pas nommée explicitement et on écrira par exemple:

```
let length = fun
| cons a b -> (+ 1 (length b))
| nil -> 0
```

On aura ainsi `length nil = 0` et `length (cons t1 t2) = (+ 1 (length t2))`.

2.2.1 Expressions

Une expression est construite selon l'une des formes suivantes:

- une variable `x`.
- un nom de fonction `f`.
- une construction $(c\ e_1 \ldots e_n)$ avec $c \in \mathcal{C}ons$ un constructeur et $e_1 \ldots e_n$ des expressions, n pouvant être nul.
- une construction $(\pi\ e_1 \ldots e_n)$ avec $\pi \in \Pi$ une primitive et $e_1 \ldots e_n$ des expressions, n pouvant être nul.
- un appel de fonction $(e_1\ e_2)$ avec e_1, e_2 des expressions.

2.2.2 Déclaration de fonctions

Une déclaration de fonction est toujours de l'une des deux forme suivantes:

- `let f` $x_1 \ldots x_n = e$ ou les x_i sont des variables et e une expression. La fonction est d'arité n.
- `let f` $x_1 \ldots x_n =$ `fun pat |...| pat` ou les x_i sont des variables et `pat` des motifs. La fonction est d'arité $n+1$. Un motif est une déclaration de la forme $c\ y_1 \ldots y_m \rightarrow e$, ou c est un constructeur, les y_j des variables, et e une expression.

La notion d'arité que j'utilise ici est très syntaxique et ne dépend que de la forme de la déclaration de la fonction. Dans la mesure ou le langage n'est pas typé, on ne peut pas définir correctement une notion d'arité qui soit sémantique. Donc, si on a les déclarations suivantes:

```
let f x y = (+ x y)
let g = (f 1)
```

alors la fonction `f` est d'arité 2 car elle a syntaxiquement à deux variables et la fonction `g` est d'arité nulle car elle n'a pas (syntaxiquement) de variable, même si (sémantiquement) on peut lui appliquer un argument.

2.2.3 Programme

Un programme est un ensemble de déclaration de fonctions. Pour un programme fonctionnel F, on appelle $D_F(\mathtt{f})$ la déclaration de la fonction `f` dans ce programme, et $A_F(\mathtt{f})$ son arité.

2.2.4 Sucre syntaxique

Pour simplifier l'écriture des programmes fonctionnels, j'utiliserai des notations standards qui viennent enrichir le langage des expressions. Toutes ces notations sont standard, je ne les rappelle que dans un souci de notation.

Termes fonctionnels: une expression de la forme `fun x ->e` représente en fait l'expression `(f x1...xn)`, où `x1...xn` sont les variables apparaissant dans l'expression `e` et où `f` est une nouvelle fonction dont la déclaration est:

```
let f x1...xn x = e
```

"Pattern matching": une expression de la forme `match e with P` représente en fait l'expression `f x1...xn e`, où `x1...xn` sont les variables apparaissant dans `P` et où `f` est une nouvelle fonction dont la déclaration est:

```
let f x1...xn = fun P
```

Définitions locales: une expression de la forme `let x = e1 in e2` représente l'expression `((fun x ->e2) e1)`.

On peut ainsi ajouter de nombreuses autres constructions syntaxiques.

2.3 Sémantique

Comme je l'ai mentionné dans l'introduction du chapitre, la sémantique de mon langage fonctionnel de référence est de type appel par valeur, sans restriction sur l'ordre dans lequel les arguments sont évalués.

2.3.1 Valeurs

Soit donc un programme F. Je commence par définir un sous-ensemble des termes qui formeront les valeurs.

Définition: Une valeur ressemble à une expression et elle est de l'une des formes suivantes:

- $(c\ v_1 \ldots v_n)$ avec $c \in \mathcal{C}ons$, $v_1 \ldots v_n$ des valeurs et $n \geq 0$.
- $(\pi\ v_1 \ldots v_n)$ avec $\pi \in \Pi$, $v_1 \ldots v_n$ des valeurs et $n \geq 0$.

- ($\mathtt{f}$ $v_1 \dots v_n$) avec $v_1 \dots v_n$ des valeurs, $\mathtt{f}$ une fonction du programme et $n \leq A_F(\mathtt{f})$. La valeur est appelée une fermeture de fonction, elle représente une application de n arguments à la fonction $\mathtt{f}$. Si $n = A_F(\mathtt{f})$, on dit que la fermeture est complète ou saturée, dans le cas contraire on dit qu'elle est partielle.

2.3.2 Environnements

Définition: Un environnement γ est une fonction associant des variables à des expression. Quand $\gamma(x)$ n'existe pas, je le note $\gamma(x) = \perp$. Enfin, l'extension $\gamma(x : v)$ donne un nouvel environnement dans lequel x est associé à v, et les autres variables y restent associées à $\gamma(y)$. En cas de confusion, c'est l'extension qui a priorité.

2.3.3 Dérivation

Je définis maintenant une relation entre expressions, dite de dérivation et notée $\rightarrow_\gamma$. La sémantique du programme sera tout simplement la restriction de $\rightarrow_\gamma$ aux valeurs.

Voyons maintenant comment est définie cette relation sur des expressions quelconques. Quand la relation n'est pas définie pour une expression e, c'est un cas particulier que je noterai $e \rightarrow_\gamma \perp$. Sinon, la relation est définie quand l'une des règles suivantes s'applique:

Définition: Règles définissant la relation $\rightarrow_\gamma$:

- une variable se dérive en la valeur qui est dans γ: $x \rightarrow_\gamma \gamma(x)$.
- Une application se dérive si son premier terme se dérive lui-même en une fermeture et que le deuxième se dérive en une valeur:

$$(e_1\ e_2) \rightarrow_\gamma v \Leftrightarrow \begin{cases} e_1 \rightarrow_\gamma (f\ v_1 \dots v_p) \\ e_2 \rightarrow_\gamma v' \\ (f\ v_1 \dots v_p\ v') \rightarrow_\gamma v \end{cases}$$

- Quand une fermeture (f $v_1 \dots v_n$) est complète, c'est à dire que $n = A_F(f)$, on la remplace par l'expression qui lui est associée dans $D_F(f)$. Simultanément, les paramètres de la fonction sont associées aux valeurs $v_1 \dots v_n$ dans l'environnement γ et on dérive le tout. Il faut en fait distinguer deux cas, suivant la forme de la déclaration:

$$(f\ v_1 \dots v_n) \to_\gamma v \Leftrightarrow$$
$$\begin{cases} D_F(f) = \texttt{let}\ f\ x_1 \dots x_n = e \\ \gamma' = \gamma(x_k : v_k)_{k \le n} \\ e \to_{\gamma'} v \end{cases}$$
ou bien
$$\begin{cases} D_F(f) = \texttt{let}\ f\ x_1 \dots x_{n-1} = \texttt{fun} \\ \quad \dots \mid \quad c\ y_1 \dots y_m \to e_c \quad \mid \dots \\ v_n = (c\ w_1 \dots w_m) \\ \gamma' = \gamma(x_k : v_k)_{k<n}(y_i : w_i)_{i \le m} \\ e_c \to_{\gamma'} v \end{cases}$$

- Pour les constructeurs, on a tout simplement:

$$(c\ e_1 \dots e_n) \to_\gamma (c\ v_1 \dots v_n) \quad \Leftrightarrow \quad (\forall i)\ e_i \to_\gamma v_i$$

- Les opérations primitives étant prédéfinies, je suppose qu'il existe une relation de réécriture notée $\triangleright$ qui les modélise (Cf. section 10.4). On aura donc:

$$(\pi\ e_1 \dots e_n) \to_\gamma v \Leftrightarrow \begin{cases} (\forall i)\ e_i \to_\gamma v_i \\ (\pi\ v_1 \dots v_n) \triangleright v \end{cases}$$

2.3.4 Sémantique

La sémantique du programme F est tout simplement la relation $\mathcal{S}$ entre valeurs telles que:

Définition: $v\mathcal{S}v' \Leftrightarrow v \to_\varepsilon v'$, où ε est l'environnement vide.

2.4 Transformations

Dans cette section, je donne un aperçu de quelques transformations définie sur des programmes fonctionnels. Cette présentation n'a pas la vocation d'être exhaustive, mais simplement d'être une illustration des méthodes utilisées.

2.4.1 Déforestation de Wadler

Cette méthode de déforestation fonctionne en introduisant de nouvelles fonctions pour représenter les compositions, et en essayant de simplifier statiquement les expressions d'un programme. Supposons que l'on cherche à éliminer la compositions des fonctions `f` et `g`, on se donne un nouveau nom de fonction `h` qui représentera cette composition. Je suppose que les déclarations de `f` et `g` sont de la forme:

$$\begin{array}{l}\texttt{let f}\ X_1 \dots X_m\ X = \\ \quad \texttt{match}\ X\ \texttt{with}\ P_1 \to E_1 \dots P_n \to E_n \\ \texttt{let g}\ Y_1 \dots Y_r = E\end{array}$$

Alors l'algorithme effectuant la déforestation, noté W, est une transformation qui modifie toute expression de la manière suivante:

$$\begin{array}{ll}
(1). & W[\![x]\!] = x \\
(2). & W[\![(c\ e_1 \dots e_n)]\!] = (c\ W[\![e_1]\!] \dots W[\![e_n]\!]) \\
(3). & W[\![(\varphi\ e_1 \dots\ e_n)]\!] = \\
 & \quad W[\![\ e[y_j{:=}W[\![e'_j]\!]][x_i{:=}W[\![e_i]\!]]\ ]\!] \\
 & \quad \text{avec } D_F(\varphi) = \texttt{let}\ \varphi\ x_1 \dots x_n = e \\
(4). & W[\![\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \dots p_n \to e_n]\!] = \\
 & \quad \texttt{match}\ x\ \texttt{with}\ p_1 \to W[\![e_1]\!] \dots p_n \to W[\![e_n]\!] \\
(5). & W[\![\texttt{match}\ (c\ e_1 \dots\ e_n)\ \texttt{with}\ \dots\ p \to e \dots]\!] = \\
 & \quad W[\![e[x_i{:=}[\![e_i]\!]]\!] \\
 & \quad \text{avec } p = c\ x_1 \dots x_n \\
(6). & W[\![\texttt{match}\ (\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \dots p_n \to e_n) \\
 & \qquad \texttt{with}\ p'_1 \to e'_1 \dots p'_m \to e'_m]\!] \\
 & \qquad = W[\![\texttt{match}\ e\ \texttt{with} \\
 & \qquad\quad p_1 \to \texttt{match}\ e_1\ \texttt{with}\ p'_1 \to e'_1 \dots p'_m \to e'_m \\
 & \qquad\quad \dots \\
 & \qquad\quad p_n \to \texttt{match}\ e_n\ \texttt{with}\ p'_1 \to e'_1 \dots p'_m \to e'_m \quad]\!] \\
(7). & W[\![(\texttt{f}\ e_1 \dots\ e_n\ (\texttt{g}\ e'_1 \dots e'_m\ e)]\!] = \\
 & \quad (\texttt{h}\ W[\![e_1]\!] \dots W[\![e_n]\!]\ W[\![e'_1]\!] \dots W[\![e'_m]\!]\ W[\![e]\!])
\end{array}$$

Je ne donne pas ici toutes les règles de calcul de $W[\![.]\!]$, d'autant plus qu'il faut en pratique modifier l'algorithme initial de Wadler pour s'assurer de sa terminaison dans le cas général (voir [44, 18], ou [5] pour des versions complètes). On effectue cette transformation$W[\![.]\!]$ sur le corps des fonctions du programme. Puis, on introduit la définition de la nouvelle fonction `h`, ainsi:

$$\begin{array}{l}\texttt{let h}\ X_1 \dots X_m\ Y_1 \dots Y_r = \\ \quad W[\![\texttt{match}\ E\ \texttt{with}\ P_1 \to E_1 \dots P_n \to E_n]\!]\end{array}$$

L'effet de la fonction $W[\![.]\!]$ est double. Tout d'abord, avec les règles (1) à (6), l'algorithme effectue ce que l'on pourrait appeler une évaluation incomplète (partielle?) du corps des fonctions: les règles qui définissent la sémantique du langage fonctionnel sont appliquée autant que possible. Le deuxième effet de l'algorithme, avec la règle (7), est de remplacer les occurrences de la composition des fonctions `f` et `g` par la fonction `h`. On effectue ainsi de nombreuses déforestations.

Exemple: prenons le programme initial suivant:

```
let map f = fun
| cons a b -> cons (f a) (map f b)
| nil -> nil

let main f g x = map f (map g x)
```

Si on veut éliminer les compositions de `map` et `map`, on obtient le programme suivant, où `m` est la fonction composée:

```
let m f g = fun
| cons a b -> cons (f (g a)) (m f g b)
| nil -> nil

let main f g x = m f g x
```

On a bien obtenu une déforestation, dans la mesure où il n'y a plus de liste intermédiaire construite.

Un cas d'échec: prenons maintenant le programme suivant :

```
let append xs ys = match xs with
| cons a b -> cons a (append b ys)
| nil -> ys

let main x y z = append (append x y) z
```

Ce programme concatène trois listes, mais construit trop de listes intermédiaires. Une meilleure version du programme est par exemple:

```
let main x y z = append x (append y z)
```

Pourtant, sur le programme initial, l'algorithme de Wadler tombe dans un cas de non terminaison, que l'on évite en marquant certains termes (méthode proposée par Wadler [44]) où en faisant des analyses statiques qui donnent une approximation de la terminaison de l'algorithme (approche proposée dans [37, 32, 11, 34, 33]). Avec ces approches plus complexes, cependant, on ne parvient toujours pas a traiter la plupart des cas de déforestation dans les accumulateurs. Je présenterai ces autres cas d'échec au chapitre 4 en parallèle avec les résultats obtenus en utilisant la sémantique équationnelle.

2.4.2 Évaluation Partielle et Spécialisation

Supposons que l'on ait une expression de la forme (`f` v e), avec `f` un nom de fonction, v une valeur particulière, e une expression quelconque. Une transformation intéressante est alors d'introduire la fonction `h` définie par:

```
let h x = f v x
```

Puis, avec un algorithme semblable au précédant, on remplace dans toutes les déclarations du programme les expressions de la forme (`f` v e) par l'expression (`h` e). On a ainsi spécialisé `f` dans le cas où son premier argument est égal à `v`. La connaissance de cette valeur se propage dans le corps de la fonction `h` en faisant apparaître des simplifications.

Essentiellement, cet algorithme peut reprendre les règles (1) à (6) de l'algorithme de Wadler, et de modifier la règle (7) pour introduire la nouvelle fonction `h` qui spécialise `f`. Encore une fois, toute la difficulté réside dans le fait de savoir si la transformation termine ou si il est pertinent de poursuivre trop loin des simplifications.

Remarque: dans ces deux transformations, il y a une idée commune. En effet, on fabrique des nouvelles fonctions qui capturent la transformation attendue, puis on utilise les règles de la sémantique du langage fonctionnel pour effectuer des simplifications tout en introduisant aux bons endroits les nouvelles fonctions. Effectuer des simplifications, c'est è dire les règles (1) à (6) de l'algorithme de Wadler, correspond littéralement à évaluer partiellement les fonctions du programme. C'est à partir de cette remarque que j'ai construit mes méthodes de déforestation, en respectant cette idée simple qu'il suffit d'introduire des nouvelles fonctions bien choisies et à étendre en conséquence un algorithme de calcul que j'appelerai dans la suite évaluation partielle (voir section 5.1).

2.4.3 Déforestation à base de *folds*

Dans les techniques de déforestation, certaines approches fonctionnent en modélisant la manière dont les données intermédiaire sont construites et consommées par les fonctions. Parmi ces méthodes, citons celle de Gill, Launchbury et Jones [10, 9] ou encore de Sheard et Fegaras [36] mettant en oeuvre une stratégie à base de *fold*. Ces méthodes sont composées de trois parties: on définit des opérateurs qui consomment des arbres (généralisés, comme les listes), puis des opérateurs qui les construisent, et enfin une règle de fusion qui simplifie les enchaînement de constructeurs et de consommateurs.

Consommateurs: Intuitivement, un *fold* est un opérateur qui représente une fonction récursive sur un arbre. Dans le cas des listes, cet opérateur est simplement la fonction suivante:

```
let fold f a = fun
  | cons e l -> f e (fold f a l)
  | nil -> a
```

C'est en effet un opérateur qui permet d'abstraire le comportement récursif d'une fonction sur les listes. Par exemple, on peut définir les fonctions classiques sur les listes de la manière suivante:

```
let length a = fold (fun x y -> 1+y) 0 a
let append a b = fold (fun x y -> (cons x y)) (nil) b a
let map f a = fold (fun x y -> (cons (f x) y)) (nil) a
let id a = fold (fun x y -> (cons x y)) (nil) a
```

Cette notion de *fold* permet ainsi d'abstraire la manière dont une fonction consomme une liste.

Constructeurs: pour modéliser la manière dont une fonction construit un arbre, on introduit la notion de *build*. La définition de l'opérateur *build* pour les listes est:

```
let build f = f (fun x y -> (cons x y)) (nil)
```

Quand une fonction `f` construit des listes, on peut l'écrire avec `build`. Prenons par exemple la fonction suivante, qui construit une liste d'entiers:

```
let inter a b =
  if a<b then (cons a (inter (a+1) b))
  else nil
```

Cette fonction peut aussi s'écrire avec `build` avec l'aide d'une fonction auxiliaire `inter'` qui capture ce qu'on appelle le schéma de récursion de la fonction `inter`. Voici ce que l'on obtient:

```
let inter' a b =
  (fun c n ->
    if a<b then (c a (inter' (a+1) b c n))
           else (n)
  )
let inter = build inter'
```

C'est une version un peu compliquée de la fonction initiale et il est en pratique difficile de programmer en utilisant des *folds*. En pratique, les méthodes de déforestation utilisant ces notions nécessitent l'éllaboration de bibliothèques contenant les versions avec *fold* et *build* de toutes les fonctions classiquement utilisées comme `length`, `append`, etc.

Déforestation: La déforestation est appliquée par une règle appelée règle de fusion, qui est donnée sous la forme suivante:

```
fold a b (build g) = g a b
```

Cette règle concrétise exactement l'idée que l'on peut éviter de construire une liste (*build*) qui est immédiatement consommée (*fold*). Prenons un exemple:

```
let main a b = length (inter a b)
```

Si on utilise les opérateurs `build` et `fold`, on trouve la définition équivalente suivante:

```
let main a b = fold (fun x y -> 1+y) (0) (build (inter' a b))
```

En appliquant la règle de fusion, on obtient la simplification suivante:

```
let main a b =  (inter' a b) (fun x y -> 1+y) (0)
```

Cette expression peut alors être simplifiée par une spécialisation et on trouve la solution finale suivante:

```
let main a b = if a<b then 1+(main (a+1) b) else 0
```

Généralisation: la technique à base de *fold* est généralisable à tout type d'arbre [20, 23]. Ce sont des opérations assez délicates dont on peut trouver une description synthétique mais complète dans la thèse de Duris [5]. Il est inutile d'en donner les détails ici. La même thèse effectue une comparaison poussée avec l'algorithme de Wadler, pour aboutir à la conclusion que les deux méthodes ont des résultats similaires.

2.4.4 Hylomorphismes

Une autre généralisation des *folds* utilise beaucoup plus en profondeur les notions de constructions et de schémas de récursion. Je veux parler ici de la notion d'hylomorphisme introduite, entres autres, par Meijer, Fokkinga et Paterson [22]. Pour généraliser les *folds*, on essaye d'abstraire non seulement le comportement des fonctions, mais aussi les données manipulée, en exprimant les deux types d'objets avec un même formalisme algébrique. On utilise pour cela les notions de foncteur et de catégorie.

Foncteurs: un foncteur est un objet construit à partir du foncteur identité id, de foncteurs constant c, des opérateurs $+$ et $\times$, et de l'opérateur de point fixe μ, et de variables de foncteur X, Y, etc. A chaque foncteur F est associé deux opérations, une sur des ensembles et une sur des fonctions. Ces opérations prennent autant de paramètres que de variables apparaissant dans le foncteur, et je noterai $\overline{X}$ le n-uplet de variables $X_1 \dots X_n$. J'utilise la notation du λ-calcul pour représenter ces opérations dans le tableau ci-dessous:

<table>
<tr><th>Foncteur</th><th>Ensemble</th><th>Fonction</th></tr>
<tr><td>c
id</td><td>$\{c\}$
$\lambda X.X$</td><td>c
$\lambda X.X$</td></tr>
<tr><td>$F_1 + F_2$</td><td>$\lambda\overline{X}.$
$\{(1,x) \mid x \in (F_1\ \overline{X})\}$
$\{(2,y) \mid y \in (F_2\ \overline{X})\}$</td><td>$\lambda\overline{X}.\lambda(k,e).$
$\texttt{if}\ k = 1\ \texttt{then}\ (F_1\ \overline{X}\ e)$
$\texttt{if}\ k = 2\ \texttt{then}\ (F_2\ \overline{X}\ e)$</td></tr>
<tr><td>$F_1 \times F_2$</td><td>$\lambda\overline{X}.\{(x,y) \mid$
$x \in (F_1\ \overline{X})$
$y \in (F_2\ \overline{X})\}$</td><td>$\lambda\overline{X}.\lambda(a,b).$
$(F_1\ \overline{X}\ a), (F_2\ \overline{X}\ b)$</td></tr>
<tr><td>μF</td><td>$\texttt{fix}\ (\lambda X.F\ X)$</td><td>$\texttt{fix}\ (\lambda X.F\ X)$</td></tr>
</table>

Hylomorphismes: avec un foncteur, on peut donc représenter dans le même formalisme un type de donnée et une fonction. Par exemple, les listes d'éléments d'un ensemble E est représenté par le foncteur $(\mu L\ 1\ E)$, où L est le foncteur suivant:

$$L = \lambda V, A, X.(V + A \times X)$$

Dans ce foncteur, V représente la liste vide, et $(A \times X)$ représente la liste formée de l'élément A et de la liste X. Avec ce même foncteur L, on peut représenter la longueur d'une liste par la fonction $\mu L\ (\lambda x.0)\ (\lambda x, y.1 + y)$.

Le but recherché avec cette notion de foncteur est d'abstraire de la même manière les schémas de récursion des fonctions (que l'on pourrait aussi appeler leur flot de contrôle) et les schémas de récursion des données. Il existe alors des théorèmes [22, 21, 40] ainsi que de nombreux algorithmes [19, 13] montrant que toute fonction récursive primitive peut être codée sous la forme d'un hylomorphisme noté $[\![\varphi, \eta, \psi]\!]_{G,F}$ qui représente la fonction suivante:

$$[\![\varphi, \eta, \psi]\!]_{G,F} = \mu(\ \lambda f.\varphi \circ (\eta \circ (F\ \ f)) \circ \psi\ )$$

Cette expression compliquée peut être vue comme une fonction récursive dont voici le schéma de calcul: Elle prend un objet a comme paramètre et le transforme grâce à ψ en un objet b qui est lui même transformé (récursivement) par la fonction $\eta \circ (F\ \ f)$ en un objet c. Il est à son tour transformé par la fonction φ en le résultat final d de la fonction. Les foncteurs F et G servent à représenter les schémas de récursions associés aux données intermédiaires b et c du calcul.

Avec cette représentation on peut coder les fonctions définies dans un langage fonctionnel, il y a même plusieurs représentations possibles pour chaque fonction. On peut alors démontrer des règles de fusions [40] qui ressemblent à la règle de fusion des *fold* et qui permettent de fusionner des hylomorphismes. Ces techniques ont été mises en oeuvre avec des résultats comparables aux autres techniques de déforestation fonctionnelles [24].

Déforestation: la notion d'hylomorphisme et leur utilisation pour la déforestation est entièrement présentée dans [5]. Il y est démontré que malgré les généralisations apportées par ce formalisme et la possibilité de l'automatiser entièrement [24], on ne parvient toujours pas a traiter le cas des fonctions avec paramètres accumulatifs qui sont déforestés avec des grammaires attribuées. Je reviendrai sur ce point au chapitre 4.

2.5 Conclusion

Les méthodes classiques de déforestation fonctionnelles ont été largement étudiées et présentent l'avantage d'être directement utilisables dans certaines situations sans recourir à des techniques trop coûteuses (comme l'algorithme de Wadler par exemple). En revanche, les tentatives de généralisation à base de notions algébriques me semblent un peu décevantes pour un objectif de déforestation (ces approches ont bon nombre d'autres applications, notemment dans la définition d'algorithmes génériques, comme en témoigne par exemple [15, 14]).

Autrement dit, les méthodes fonctionnelles ne suffisent plus dès que l'on doit déforester des compositions de fonctions utilisant des accumulateurs, comme `rev`, `append`, etc. Pourtant, les compositions mettant en jeu ces fonctions sont facilement déforestées avec des grammaire attribuées. C'est pourquoi il est important de définir un formalisme non fonctionnel pour y unifier ces différentes méthodes de déforestation.

Chapitre 3

Sémantique Équationnelle

3.1 Introduction

Dans la mesure où les méthodes fonctionnelles de déforestation ne suffisent pas à traiter toutes les situations que l'on voudrait, il faut s'inspirer d'un autre formalisme où la déforestation donne les résultats attendus: les grammaires attribuées. On pourrait imaginer de traduire les programmes fonctionnels en grammaires attribuées, de les déforester, puis de traduire le résultat en un nouveau programme fonctionnel. Prenons juste un exemple illustrant les résultats obtenus par cette méthode:

```
let flat l = fun
  | node a b -> (flat l a) b
  | leaf n -> (cons n l)

let rev h x = match x with
  | cons a b -> rev (cons a h) b
  | nil -> h

let main t = rev nil t
```

La fonction `rev` inverse une liste avec un paramètre accumulatif. La fonction `flat` calcule, également avec un accumlateur, la liste des feuilles d'un arbre binaire de gauche à droite. Enfin, la fonction `main` combine les deux précédentes et donne la liste des feuilles de l'arbre `t`, mais de droite à gauche cette fois ci. Ces deux fonctions sont linéaires et peuvent être codées avec des grammaires attribuées, la composition des deux (la fonction `main`) est alors déforestée en une nouvelle grammaire attribuée qui est traduite en le programme fonctionnel suivant:

```
let rflat l = fun
  | node a b -> (rflat l b) a
  | leaf n -> (cons n l)

let main t = rflat nil t
```

Le résultat utilise une fonction presque identique à `flat` mais qui traverse les branches de l'arbre binaire dans le sens opposé, sans construire de liste intermédiaire. L'utilisation des grammaires attribuées comme forme intermédiaire pour la déforestation semble donc être une bonne voie.

Malheureusement, cette solution idéalisée n'est pas réalisable, en raison du manque de pouvoir d'expression des grammaire attribuées. En effet, une grammaire attribuée ne peut modéliser que des algorithmes linéaires. On ne peut donc pas toujours traduire une fonction en une grammaire attribuée. Autre conséquence, la composition de deux algorithmes linéaires n'étant pas toujours un algorithme linéaire, la déforestation dans les grammaires attribuées peut échouer [7, 31]. Par exemple, prenons le programme suivant, qui calcule la puissance deux d'un entier de Peano:

```
let exptree = fun
  | succ n -> let t = exptree n in (node t t)
  | zero -> (leaf)

let size h = fun
  | (node a b) -> size (size h a) b
  | (leaf) -> (succ h)

let main n = size zero (exptree n)
```

Ecrit ainsi, le calcul de la puissance deux n'est pas très efficace car on contruit un arbre binaire intermédiaire qui contient autant de feuilles et de noeuds que le résultat final. Or, la plupart des méthodes de déforestation fonctionnelle permettent d'obtenir le résultat déforesté suivant:

```
let exp h = fun
  | succ n -> exp (exp h a) b
  | zero -> (succ h)

let main n = exp zero n
```

Pour pouvoir introduire les techniques de déforestation des grammaires attribuées dans la programmation fonctionnelle, il fallait donc imaginer un formalisme qui regroupe simultanément les avantages des deux formalismes, en espérant ainsi pouvoir traiter plus de cas comme le précédent. Intuitivement, on peut dire que les grammaires attribuées ont un pouvoir d'expression limité en raison de leurs restrictions: on ne peut pas créer d'équations entre des attributs d'un arbre qui serait construit dynamiquement. Pour étendre le pouvoir d'expression des grammaires attribuées, il faut donc y introduire une notion de construction dynamique [28, 29]. Mais ce n'est pas suffisant car les grammaires attribuées classiques sont fortement typées et incorporer l'ordre supérieur des langages fonctionnels requiert un système de type perfectionné, ou bien l'absence de type (comme dans le λ-calcul non typé). C'est sur ces idées que j'ai construit la sémantique équationnelle.

Je ne ferai donc pas de présentation des grammaires attribuées dans la mesure où elles ne sont que des cas particuliers du formalisme que je présente ici.

Concernant l'organisation du présent chapitre, je commence par définir formellement ce que j'appelle un programme équationnel (section 3.2), suivi de quelques exemples introductifs (section 3.3). Ensuite, je présenterai quelques objectifs et motivations (section 3.4), puis viendra la définition formelle de la sémantique des programmes équationnels, que j'appelle naturellement sémantique équationnelle. Ces définitions formelles se veulent simples et concises.

Viennent ensuite des sections qui sont des introductions aux notions et aux outils que j'utilise pour produire des transformations de programme. Les problèmes techniques soulevés sont exposés successivement avec des idées intuitives pour leur solution, accompagnées de notations utiles. Chaque point relevé fera l'objet d'un chapitre à part dans la suite de ma thèse et je terminerai ce chapitre de présentation par la section 3.8 qui est une sorte de notice de lecture.

3.2 Programmes Équationnels

Dans cette section, je présente les définitions nécessaires à la sémantique équationnelle. On pourra donc se servir de cette section comme d'un catalogue ou d'un glossaire. Notons que ces définitions sont mutuellement dépendantes et qu'il faut les comprendre globalement.

3.2.1 Termes

J'emprunte librement cette notion à la programmation fonctionnelle.

Définition: un terme t est construit à partir d'un symbole f et de termes $t_1 \ldots t_n$, n pouvant être nul. Il est noté $t = (f\ t_1\ t_2 \ldots t_n)$. Le symbole f peut être un constructeur $c \in \mathcal{C}ons$, une primitive $\pi \in \mathcal{P}rim$ ou une variable $x \in \mathcal{V}ars$ (dans ce dernier cas n doit être nul). L'ensemble des termes est noté $\mathcal{T}erms$.

Définition: je noterai $\mathcal{V}ars(t)$ l'ensembles des variables qui sont utilisées pour construire le terme t. Par extension, $\mathcal{V}ars(A)$ est l'ensemble des variables qui apparaissent dans A, que ce soit un terme, une équation, un programme, etc.

Définition: une valeur v est un terme sans variable, c'est à dire tel que $\mathcal{V}ars(v) = \emptyset$.

3.2.2 Variables

Les variables sont utilisées pour désigner d'autres termes. La manière de former les noms de variables est canonique et dépend de ce que représente chaque variable.

Définition: une variable est de l'une des formes suivantes:

- $x.k$, avec k un entier et x une autre variable. Elle représente le k-ième sous-terme du terme représenté par k.
- $x.L_k$, avec k un entier et x une autre variable. Elle représente la k-ième variable locale associée à x, une notion très utile pour les constructions dynamiques ou non-linéaires (voir les sections 3.3.4 et 3.3.5).
- $x.a$, où a est un attribut et x une autre variable. Elle représente la valeur de l'attribut a associée à la variable x.
- la variable universelle α. Elle représente une variable universellement quantifiée.

A noter enfin que si $x \in \mathcal{V}ars(t)$, alors les sous-variables qui constituent x n'appartiennent pas à $\mathcal{V}ars(t)$. Ainsi, avec $f \in \mathcal{C}ons$, on aura par exemple $x \in \mathcal{V}ars(f\ x\ y)$, mais $x \notin \mathcal{V}ars(f\ x.1\ x.2.3)$. Je reviendrai sur cette distinction à propos des substitutions (voir section 3.5).

3.2.3 Attributs

Les attributs sont des étiquettes servant à représenter les résultats et les paramètres d'un calcul mené sur un terme.

Définition: je définis deux ensembles d'attributs, $\mathcal{R}es$ pour les résultats, et $\mathcal{P}rm$ pour les paramètres. L'ensemble $\mathcal{A}tt$ de tous les attributs est la réunion (disjointe) de $\mathcal{P}rm$ et $\mathcal{R}es$.

Avec $r \in \mathcal{R}es$ et $p \in \mathcal{P}rm$, je dirai par exemple que $x.r$ est le résultat du calcul de r sur le terme représenté par la variable x et avec pour paramètre p le terme représenté par la variable $x.p$.

La notion d'attribut que je propose ici est exactement la même que pour les grammaires attribuées.

3.2.4 Équations & Systèmes

Mes équations sont toujours orientées, avec une partie gauche restreinte à être une variable et une partie droite pouvant être un terme quelconque.

Définition: une équation est de la forme $x = t$ avec x une variable et t un terme.

Définition: un système est un ensemble d'équations.

3.2.5 Programme Équationnel

Un programme sera constitué de formules logiques permettant de produire de nouvelles équations. Mais je me limite toujours à des formules logiques de la forme:

$$(\forall x)(\forall t_1 \ldots t_m \in \mathcal{T}erms) \quad x = (c\ t_1 \ldots t_m) \Rightarrow y = t$$

avec $c \in \mathcal{C}ons$. On dira d'ailleurs que l'équation $y = t$ est associée ou définie sur le constructeur c. En pratique j'utilise une notation abrégée à propos de la variable universellement quantifiée x, que je note α.

Définition: une équation de programme est de la forme $c \rightarrow y = t$, avec c un constructeur, et $y = t$ une équation.

Définition: un programme équationnel, noté $\mathcal{P}rg$, est un ensemble d'équations de programme.

Je suppose pour simplifier que les ensembles de constructeurs, de variables et d'attributs sont communs à tous les programmes considérés, de sorte que l'on peut librement les réunir par union ensembliste. Je m'oppose donc ici à la vision classique des grammaires attribuées ou chaque grammaire (programme) doit définir ses propres attributs, constructeurs, etc.

3.2.6 Grammaire BNF

Pour résumer ce qui vient d'être dit à propos de la syntaxe des programmes équationnels, je propose cette petite grammaire:

$$\boxed{\begin{array}{lcl}
\mathcal{P}rg & ::= & \{c \rightarrow eq \ldots\} \\
eq & ::= & x = t \\
x & ::= & \alpha \mid x.k \mid x.a \mid x.L_k \\
t & ::= & x \\
 & \mid & (c\ t_1 \ldots t_n) \\
 & \mid & (\pi\ t_1 \ldots t_n) \\
\Sigma & ::= & \{eq \ldots\}
\end{array}}$$

3.3 Exemples

3.3.1 Représentation des termes

Dans un premier temps, je vais faire quelques remarques sur les systèmes d'équations représentant des termes. Prenons la liste suivante:

$$(cons\ 5\ (cons\ 6\ nil))$$

Soit x une variable (équationnelle), alors on peut former le système d'équation suivant:

$$\{x = (cons\ 5\ (cons\ 6\ nil))$$

Comme je l'ai annoncé à la section 2.2 les variables de la forme $x.k$ sont sensées représenter le k-ième sous-terme du terme représenté par la variable x. Donc, ici, on aura:

$$\left\{ \begin{array}{ccl} x & = & (cons\ 5\ (cons\ 6\ nil)) \\ x.1 & = & 5 \\ x.2 & = & (cons\ 6\ nil) \\ x.2.1 & = & 6 \\ x.2.2 & = & (nil) \end{array} \right\}$$

3.3.2 Codage d'une fonction sans accumulateur

Une fois que l'on sait coder les termes et leurs sous-termes, on peut écrire des programmes. Prenons pour exemple la longueur d'une liste. Avec un langage fonctionnel on écrit le programme suivant:

```
let length = fun
| cons a b -> (+ 1 (length b))
| nil -> 0
```

L'équivalent en programme équationnel consiste à définir un attribut *length*. Ainsi, pour toute variable x représentant une liste, alors la variable $x.length$ représentera la longueur de cette liste. Avec cette idée, la valeur de chaque attribut *length* est déterminée par des équations que l'on peut écrire ainsi:

$$\left\{ \begin{array}{ccl} (\forall x, a, b) & x = (cons\ a\ b) & \Rightarrow x.length = (+\ 1\ x.2.length) \\ (\forall x) & x = (nil) & \Rightarrow x.length = 0 \end{array} \right\}$$

La notation proposée à la section 2.2 permet de simplifier l'écriture de ces propriétés vérifiées par l'attribut *length*. Ainsi, on écrira le programme équationnel suivant:

$$\mathcal{P}rg = \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.length = (+\ 1\ \alpha.2.length) \\ nil & \rightarrow & \alpha.length = 0 \end{array} \right\}$$

Le lecteur remarquera qu'il y a beaucoup de ressemblance entre $\mathcal{P}rg$ et le programme fonctionnel initial. Maintenant, voyons comment on peut utiliser ces propriétés pour effectivement calculer la longueur d'une liste. Partons du système initial suivant:

$$\Sigma_0 = \{x = (cons\ 5\ (cons\ 6\ nil))\}$$

Avec les conventions sur les noms de variables (voir l'exemple 1), on peut obtenir le système d'équations suivant:

$$\Sigma_1 = \left\{ \begin{array}{ccl} x & = & (cons\ 5\ (cons\ 6\ nil)) \\ x.1 & = & 5 \\ x.2 & = & (cons\ 6\ nil) \\ x.2.1 & = & 6 \\ x.2.2 & = & (nil) \end{array} \right\}$$

Si maintenant j'utilise les propriétés énnoncées par le programme $\mathcal{P}rg$, on obtient le système d'équations suivant:

$$\Sigma_2 = \left\{ \begin{array}{ccc} x.length & = & (+\ 1\ x.2.length) \\ x.2.length & = & (+\ 1\ x.2.2.length) \\ x.2.2.length & = & 0 \end{array} \right\}$$

Ce système est triangulaire et donc se résoud très facilement, la solution finale étant bien entendu:

$$\Sigma_3 = \left\{ \begin{array}{ccc} x.length & = & 2 \\ x.2.length & = & 1 \\ x.2.2.length & = & 0 \end{array} \right\}$$

Ainsi, à partir de l'équation $x = (cons\ 5\ (cons\ 6\ nil))$, on a obtenu les valeurs de la variable $x.length$ grâce aux opérations suivantes:

1.	grâce aux conventions sur les noms de variables, obtenir le système Σ_1.
2.	appliquer les propriétés $\mathcal{P}rg$ pour obtenir le système Σ_2.
3.	résoudre Σ_2 pour obtenir Σ_3.

La sémantique équationnelle va permettre de formaliser cette notion intuitive de calcul par résolution d'un système d'équations. Ainsi, je définirai une relation notée $\Rightarrow_{\mathcal{P}rg}$ dont les propriétés resemblent beaucoup à l'implication logique et avec laquelle on pourra écrire:

$$\Sigma_0 \Rightarrow_{\mathcal{P}rg} \Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma_2 \Rightarrow_{\mathcal{P}rg} \Sigma_3$$

3.3.3 Codage d'une fonction avec accumulateur

Dans l'exemple précédant, la fonction codée n'avait qu'un seul paramètre: la liste dont on veut calculer la longueur. Si maintenant on veut coder l'inversion avec accumulateur, il faut pouvoir représenter un paramètre supplémentaire. Je rappelle tout d'abord la fonction d'inversion de liste:

```
let rev h = fun
  cons a b -> rev (cons a h) b
| nil -> h
```

Prenons encore une fois la liste x de l'exemple précédent. On peut décomposer l'exécution de la fonction `rev` sur cette liste de la manière suivante. Initialement, on a le terme suivant:

```
rev nil (cons 5 (cons 6 nil))
```

Ensuite ce terme devient successivement:

```
rev nil (cons 5 (cons 6 nil))
= rev (cons 5 nil) (cons 6 nil)
= rev (cons 6 (cons 5 nil)) nil
= (cons 6 (cons 5 nil))
```

Pour coder ce calcul avec un programme équationnel, je vais utiliser deux attributs. Le premier, *rev*, est un attribut calculant le résultat de l'inversion, et on aura donc $rev \in \mathcal{R}es$. Le second attribut, *rev_h*, représentera le paramètre accumulatif de la fonction, et on aura $rev\_h \in \mathcal{P}rm$.

Avec ces notations, les attributs *rev* et *rev_h* vérifient les propriétés suivantes que je note par un programme équationnel $\mathcal{P}rg$:

$$\mathcal{P}rg = \left\{ \begin{array}{lcl} cons & \rightarrow & \alpha.rev = \alpha.2.rev \\ cons & \rightarrow & \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ nil & \rightarrow & \alpha.rev = \alpha.rev\_h \end{array} \right\}$$

On peut faire le parallèle avec le programme fonctionnel `rev`: là où on fait un appel récursif à la fonction `rev`, on utilise l'attribut *rev*, et l'on se sert de l'attribut *rev_h* pour propager la valeur du paramètre `h`.

Si on veut calculer l'inversion d'une liste avec ce programme équationnel, on peut considérer le système initial suivant:

$$\Sigma_0 = \left\{ \begin{array}{ccl} x & = & (cons\ 5\ (cons\ 6\ nil)) \\ x.rev\_h & = & (nil) \end{array} \right\}$$

Pour trouver les valeurs de la variable $x.rev$, on peut alors appliquer l'algorithme de résolution suivant:

1.	avec les conventions sur les noms de variables, obtenir un système Σ_1.
2.	appliquer les propriétés $\mathcal{P}rg$ sur Σ_1 pour obtenir un autre système Σ_2.
3.	résoudre Σ_2 pour obtenir la valeur de $x.rev$

Voici les systèmes Σ_1 et Σ_2 ainsi obtenus:

$$\Sigma_1 = \left\{ \begin{array}{ccl} x & = & (cons\ 5\ (cons\ 6\ nil)) \\ x.1 & = & 5 \\ x.2 & = & (cons\ 6\ nil) \\ x.2.1 & = & 6 \\ x.2.2 & = & (nil) \end{array} \right\}$$

$$\Sigma_2 = \left\{ \begin{array}{ccl} x.rev & = & x.2.rev \\ x.2.rev\_h & = & (cons\ x.1\ x.rev\_h) \\ x.2.rev & = & x.2.2.rev \\ x.2.2.rev\_h & = & (cons\ x.2.1\ x.2.rev\_h) \\ x.2.2.rev & = & x.2.2.rev\_h \end{array} \right\}$$

Finalement, on peut résoudre le système Σ_2 car il est triangulaire, et on obtient le système final suivant:

$$\Sigma_3 = \left\{ \begin{array}{ccl} x.rev\_h & = & (nil) \\ x.2.rev\_h & = & (cons\ 5\ nil) \\ x.2.2.rev\_h & = & (cons\ 6\ (cons\ 5\ nil) \\ x.rev & = & (cons\ 6\ (cons\ 5\ nil) \end{array} \right\}$$

3.3.4 Représentation d'une fonction

Jusqu'à présent, je n'ai représenté que les calculs effectués par les fonctions ainsi que les valeurs manipulées par ces fonctions. Mais si l'on veut coder des fonctions d'ordre supérieur, il faut pouvoir représenter les fonctions elles-mêmes ainsi que l'opération qui consiste à appliquer une fonction à un argument.

Pour effectuer ce dodage avec un programme équationnel, il suffit de choisir deux attributs. Le premier, p sera un attribut qui représentera le paramètre d'une fonction, tandis que le second, r, représentera le résultat d'une fonction. La fonction elle même sera représentée par un constructeur. Par exemple, si la fonction `length` sera codée par le constructeur (*length*), et on aura:

$$\left\{ \begin{array}{lcl} length & \to & \alpha.r = \alpha.L_1.length \\ length & \to & \alpha.L_1 = \alpha.p \end{array} \right\}$$

On voit sur cet exemple l'apparition d'une variable locale $\alpha.L_1$. Cette variable locale permet tout simplement de recopier le paramètre de la fonction $\alpha.p$ dans une variable nouvelle (à condition que L_1 ne soit pas utilisée ailleurs dans le programme). Puis la longueur de la liste représentée par $\alpha.L_1$ est calculée par $\alpha.L_1.length$, et elle est finalement recopiée dans $\alpha.r$. Par exemple, si je note `l` la liste de l'exemple précédent, on aura:

$$\left\{ \begin{array}{l} y = (length) \\ y.p = \mathtt{t} \end{array} \right\} \Rightarrow_{\mathcal{P}rg} \left\{ \begin{array}{l} y.r = y.L_1.length \\ y.L_1 = \mathtt{t} \end{array} \right\} \Rightarrow_{\mathcal{P}rg} \left\{ \begin{array}{l} y.L_1.length = 2 \\ y.r = 2 \end{array} \right\}$$

Avec cette idée on peut aussi représenter une fonction dont la récursion n'est pas dirigée par des constructeurs (comme dans le cas d'une liste), mais par des primitives. Par exemple, considérons le calcul de la fonction factorielle:

```
let fact x =
  if (< 1 x) then (* x (fact (- x 1)))
  else 1
```

Je commence par éliminner la structure conditionnelle (qui peut être vue comme du sucre syntaxique):

```
let fact x = f x (< 1 x)
let f x = fun
  true -> (* x (fact (- x 1)))
| false -> 1
```

Je peux modéliser ce programme fonctionnel par le programme équationnel suivant:

$$\mathcal{P}rg = \left\{ \begin{array}{lcl} \mathit{fact} & \rightarrow & \alpha.r = \alpha.L_1.f \\ \text{"} & \rightarrow & \alpha.L_1 = (<\ 1\ \alpha.p) \\ \text{"} & \rightarrow & \alpha.L_1.f\_x = \alpha.p \\ \mathit{true} & \rightarrow & \alpha.f = (*\ \alpha.f\_x\ \alpha.L_2.r) \\ \text{"} & \rightarrow & \alpha.L_2 = (\mathit{fact}) \\ \text{"} & \rightarrow & \alpha.L_2.p = (-\ \alpha.f\_x\ 1) \\ \mathit{false} & \rightarrow & \alpha.f = 1 \end{array} \right\}$$

Par rapport aux deux premiers exemples, l'algorithme de résolution sera un peu plus complexe, car il faudra répéter plusieurs fois le cycle d'application et de résolution des propriétés du programme $\mathcal{P}rg$. Ainsi, prenons le système initial suivant:

$$\Sigma_0 = \left\{ \begin{array}{ccc} x & = & (\mathit{fact}) \\ x.p & = & 3 \end{array} \right\}$$

Après une première application des équations du programme et une résolution de système, on obtient alors le systeme Σ_1:

$$\Sigma_1 = \left\{ \begin{array}{ccl} x.r & = & x.L_1.f \\ x.L_1 & = & (<\ 1\ x.p) \\ x.L_1 & = & (<\ 1\ 3) \\ x.L_1 & = & (\mathit{true}) \\ x.L_1.f\_x & = & x.p \\ x.L_1.f\_x & = & 3 \end{array} \right\}$$

On a pas encore la valeur de la variable $x.r$, mais on peut répéter l'application du programme sur la variable $x.L_1$ afin de calculer la variable $x.L_1.f$, ce qui donne après résolution le systeme Σ_2 (je ne donne pas toutes les équations à chaque étape pour des raisons de lisibilité):

$$\Sigma_2 = \left\{ \begin{array}{ccl} x.L_1 & = & \mathit{true} \\ x.L_1.f & = & (*\ 3\ x.L_1.L_2.r) \\ x.L_1.L_2 & = & (\mathit{fact}) \\ x.L_1.L_2.p & = & 2 \end{array} \right\}$$

Si on poursuit la résolution sur la variable $x.L_1.L_2$ pour calculer $x.L_1.L_2.r$, on aboutit au systéme suivant:

$$\Sigma_3 = \left\{ \begin{array}{ccl} x.L_1.L_2.r & = & x.L_1.L_2.L_1.r \\ x.L_1.L_2.L_1 & = & (<\ 1\ 2) \\ x.L_1.L_2.L_1 & = & (\mathit{true}) \\ x.L_1.L_2.L_1.p & = & 2 \end{array} \right\}$$

Après encore une itération du procédé (pour le dernier appel récursif) on obtient la résolution complète qui fournit l'équation $x.r = 6$.

3.3.5 Fonctions non-linéaires

Enfin, pour terminer avec cette série d'exemples, je signale que les variables locales sont très utiles dans les récursions non-linéaires, car elles permettent de distinguer deux appels d'une même fonction sur le même terme, mais avec des paramètres différents. Par exemple, considérons le programme fonctionnel suivant:

```
let exp h = fun
| succ b -> (exp (exp h b) b)
| zero -> (succ h)
```

On ne peut pas traduire ce programme en une grammaire attribuée dans la mesure où il n'est pas linéaire. Le problème principal vient du fait que l'on utilise un double appels récursifs sur le sous-arbre `b`, avec des paramètres différents, ce que l'on ne peut pas représenter en grammaire attribuées classiques. Grâce à la notion de variables locales, on peut cepedant écrire ce programme dans le fomalisme équationnel:

$$\left\{\begin{array}{ccl} succ & \rightarrow & \alpha.exp = \alpha.L_1.exp \\ " & \rightarrow & \alpha.L_1.exp\_h = \alpha.L_2.exp \\ " & \rightarrow & \alpha.L_2.exp\_h = \alpha.exp\_h \\ " & \rightarrow & \alpha.L_1 = \alpha.1 \\ " & \rightarrow & \alpha.L_2 = \alpha.1 \\ zero & \rightarrow & \alpha.exp = (succ\ \alpha.exp\_h) \end{array}\right\}$$

Les deux appels de la fonction `rev` sont ici dissociés par les variables locales construites avec L_1 et L_2 et par leurs attributs qui sont associés à des expressions différentes.

3.3.6 Résumé

Ce tour d'horizon, sous forme d'exemples, de ce à quoi peut ressembler un programme représenté par des équations est terminé. Convaincu que la notion de variable locale me permettrait effectivement d'étendre de façon homogène et simple le formalisme restreint des grammaires attribuées, il ne me reste plus qu'à définir plus précisément la sémantique d'un programme équationnel.

3.4 Motivations

Avant de commencer la définition de la sémantique équationnelle, je voudrais présenter, sous forme de commentaires, en quoi les programmes fonctionnels ne sont pas des grammaires attribuées.

Absence de typage: aucun système de type ne vient contrôler la cohérence des programmes équationnels. D'ailleurs, c'est bien mieux ainsi et cela permet d'écrire des programmes qui sont "valides" mais qu'il est difficile de typer. Ou plus exactement, cela permet d'englober tous les systèmes de typage possibles. En pratique cependant, cela rend impossible l'écriture de gros programmes. Mais je signale que la sémantique équationnelle n'est pas un formalisme destiné à la programmation, mais plutôt à la représentation de programmes écrits avec d'autres langages de programmation. En tant que forme intermédiaire d'un programme bien typé – quelqu'en soit le langage et le système de typage – un programme équationnel n'a pas besoin de typage. Je signale d'ailleurs que même un système de types polymorphes ne permettrait pas de typer correctement les attributs p et r des exemples de la section 3.3.4.

Non-déterminisme: rien n'interdit dans un programme équationnel que deux propriétés donnent deux équations ayant la même partie gauche. On a donc *à priori* plusieurs équations qui définissent une même variable. Loin d'être un problème, c'est au contraire une chance qui permet un nouveau point de vue sur la notion de transformation de programme qui sera ébauché à la section 3.6. Bien évidemment, on a parfois besoin de retrouver du déterminisme en choisissant pour chaque variable une seule équation et ce sera l'objet du chapitre 8.

Constructions dynamiques: les variables locales (celles de la forme $\alpha.L_n$) permettent de spécifier des constructions dynamiques, notion absente des grammaires attribuées et restreignant beaucoup leur pouvoir d'expression. Introduire de telles constructions est une démarche classique que l'on retrouve notamment dans [28]. Avec l'absence de typage, je trouve que leur spécification par des variables locales est particulièrement simple. Mais surtout, elle me permet d'avoir une cohérence entre les calculs statiques (ou linéaires) d'une part et dynamiques (ou non-linéaires) d'autre part. Cette cohérence est alors bien commode pour simplifier les transformations de programmes équationnels.

Une sémantique non-opérationnelle: les grammaires attribuées ont rarement eu d'autre sémantique que leurs "séquences de visites" ou leurs "graphes de dépendance"[26, 16, 31]. C'est un type de sémantique très lourd qui est particulièrement peu commode pour exprimer et certifier des transformations de programme. De plus, c'est une sémantique qui dépend fortement d'un style ou d'un modèle de calcul (déterministe, fonctionnel, impératif, avec ou sans effets de bords, etc.). Je propose au contraire une sémantique purement mathématique permettant de rendre indépendants la certification d'une transformation de programme d'une stratégie de résolution d'un programme équationnel.

3.5 Sémantique Équationnelle

Je définis ici la sémantique d'un programme équationnel $\mathcal{P}rg$. Intuitivement, cette sémantique n'est qu'une formalisation des algorithmes de résolution esquissés dans la section 3.3. L'idée générale est de partir d'un système Σ_{init} représentant les données du calcul à effectuer, et de parvenir petit à petit à un système Σ_{final} représentant les résultats de ce calcul. Je construirai le second système en ajoutant au premier des équations de telle sorte que les propriétés du programme $\mathcal{P}rg$ soient respectées.

La relation qui lie les systèmes Σ_{init} et Σ_{final} ressemble beaucoup à l'implication logique, ce qui m'incitera à introduire la notation $\Sigma_{init} \Rightarrow_{\mathcal{P}rg} \Sigma_{final}$.

Notations: j'adopte dès à présent les notations de la section 10.1 relative aux relations.

3.5.1 Substitutions

Deux types de substitutions me sont nécessaires. La raison provient du fait que l'on peut voir une variable soit comme un terme, c'est à dire sans sous-terme puisque d'arité nulle, soit comme une variable pouvant être construite avec des sous-variables. Par exemple, la variable $\alpha.2.3.a$ est construite à partir de la variable $\alpha.2.3$, mais le terme $\alpha.2.3.a$ n'est pas construit à partir du sous-terme $\alpha.2.3$.

Substitution de terme: dans un système Σ, on trouve des équations de la forme $x = t$. L'une des significations intuitives d'une telle équation est que l'on peut substituer le terme x par le terme t et réciproquement. Je définis donc une relation de substitution sur les termes, paramétrée par un système.

Définition 3.5.1 (Substitution de Termes)
Soit Σ un système. Je note $\triangleright_\Sigma$ la relation entre termes définie par:

$$\begin{array}{rcl} x \triangleright_\Sigma t & \Leftrightarrow & x = t \in \Sigma \\ (f\ t_1 \ldots t_n) \triangleright_\Sigma (f\ t'_1 \ldots\ t'_n) & \Leftrightarrow & (\exists! j) t_j \triangleright_\Sigma t'_j \quad (\forall i \neq j)\ t'_i = t_i \end{array}$$

avec $f \in \mathcal{C}ons \cup \mathcal{P}rim$.

La relation $\triangleright_\Sigma$ ne permet de substituer qu'une seule occurence d'une unique variable dans un terme. Ceci pour éviter tout problème et pour tenir compte du fait que des termes différents peuvent être reliés à la même variable dans le système Σ (conformément au non-déterminisme décrit à la section 3.4). Pour définir la relation réciproque, j'introduis les notations $t_1 \triangleleft_\Sigma t_2 \Leftrightarrow t_2 \triangleright_\Sigma t_1$, et enfin $\triangleleft\!\triangleright_\Sigma = \triangleleft_\Sigma + \triangleright_\Sigma$.

Substitution du α: je note $t[x]$ le remplacement de toute occurence de la variable α dans le terme t par la variable x. Je précise que, à la différence des précédentes substitutions, toutes les occurences de α doivent être remplacées simultanément.

Définition 3.5.2 (Substitution du α)
Etant donné un terme t et une variable x, l'opération $t[x]$ est définie par:

$$
\begin{array}{lcl}
\alpha[x] & = & x \\
y.k[x] & = & y[x].k \\
y.L_k[x] & = & y[x].L_k \\
y.a[x] & = & y[x].a \\
(f\ t_1 \dots t_n)[x] & = & (f\ t_1[x] \dots t_n[x])
\end{array}
$$

avec $f \in \mathcal{C}ons \cup \mathcal{P}rim$. Par extension, on a aussi pour tout système Σ:

$$\Sigma[x] = \{y[x] = t[x] \mid y = t \in \Sigma\}$$

Cette substitution intervient pour instancier une équation de programme $c \to y = t$ sur une variable x telle que $x = (c \dots)$. On fabrique alors l'équation définie par $y[x] = t[x]$, comme dans les exemples de la section 3.3.

3.5.2 Dérivation

Dans cette section, je définis une relation de dérivation notée $\to_{\mathcal{P}rg}$ qui définit exactement comment on fabrique de nouvelles équations à partir d'un système initial et du programme équationnel $\mathcal{P}rg$.

Définition: pour tous systèmes Σ et Σ', on a $\Sigma \to_{\mathcal{P}rg} \Sigma'$ si et seulement si l'un des cas suivants se présente:

Sous-termes: on a $x = (c\ t_1 \dots t_n) \in \Sigma$ avec $c \in \mathcal{C}ons$ et $\Sigma' = \Sigma \cup \{x.k = t_k\}$. On restreint c à être un constructeur car les primitives sont gérées différemment.

Substitutions: on a $x = t \in \Sigma$ et $\Sigma' = \Sigma \cup \{x = t'\}$ avec $t \lhd\!\rhd_\Sigma t'$.

Primitives: on a $x = t \in \Sigma$, $t \rhd t'$, et $\Sigma' = \Sigma \cup \{x = t'\}$. Je rappelle que $(\rhd)$ est une relation de réécriture sur les primitives.

Programme: on a $x = (c\ t_1 \dots t_n) \in \Sigma$, une équation de programme $c \to y = t \in \mathcal{P}rg$ et $\Sigma' = \Sigma \cup \{y[x] = t[x]\}$

Le dernier cas est le plus important et c'est le seul qui soit directement lié au programme $\mathcal{P}rg$.

3.5.3 Dérivation Totale

La sémantique la plus simple que l'on puisse imaginer pour un programme $\mathcal{P}rg$ est de collecter toutes les équations qui sont obtenues par la relation $(\to^*_{\mathcal{P}rg})$.

Définition 3.5.3 (Sémantique totale)
Étant donné le programme $\mathcal{P}rg$, sa sémantique totale est la fonction $[\![\mathcal{P}rg]\!]$ qui, à tout système Σ, associe le système définit par:

$$[\![\mathcal{P}rg]\!](\Sigma) = \bigcup\{\Sigma' \mid \Sigma \to^*_{\mathcal{P}rg} \Sigma'\}$$

J'appelerai aussi ce système la dérivation totale *à partir de Σ.*

Cette sémantique n'est pas très intéressante, car on ne peut pas s'en servir pour des transformations! En effet, le système $[\![\mathcal{P}rg]\!](\Sigma)$ contient beaucoup trop d'informations. C'est en quelque sorte la trace de toutes les étapes intermédiaires de l'exécution du programme $\mathcal{P}rg$ sur les données Σ. Or, mon but étant précisément de modifier cette exécution, cette notion de sémantique ne peut pas être retenue.

Elle a cependant un intérêt, qui est à mes yeux celui de montrer de manière très simple ce que calcule un programme, à savoir un ensemble d'équations. La sémantique équationnelle ne sera alors qu'un moyen d'escamoter de la sémantique totale les équations jugées trop encombrantes.

3.5.4 Dérivation Partielle

Je commence donc par introduire la relation $\Rightarrow_{\mathcal{P}rg}$ qui n'est qu'une souspartie de la relation $\rightarrow^*_{\mathcal{P}rg}$. Intuitivement, la relation $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$ signifie que la dérivation totale à partir de Σ fournit au moins les équations du système Σ'.

Définition 3.5.4 (Dérivation Partielle)
Pour tous systèmes Σ et Σ', on a:

$$\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma' \Leftrightarrow \Sigma' \subset [\![\mathcal{P}rg]\!](\Sigma)$$

Il est équivalent d'utiliser la définition suivante:

$$\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma' \Leftrightarrow (\exists \Sigma") \quad \Sigma \rightarrow^*_{\mathcal{P}rg} \Sigma" \text{ et } \Sigma" \supset \Sigma'$$

Dans les exemples de la section 3.3, j'ai fait une analogie entre $\Rightarrow_{\mathcal{P}rg}$ et l'implication logique. Cette comparaison est justifiée par le fait que les relations $\Rightarrow_{\mathcal{P}rg}$ et $\rightarrow_{\mathcal{P}rg}$ vérifient de nombreuses propriétés de monotonie qui font qu'on peut la manipuler exactement comme l'implication, la conjonction ("et" logique) étant remplacée par l'union de systèmes. Plus précisément, en utilisant les notions abordées dans la section 10.3, la relation $\Rightarrow_{\mathcal{P}rg}$ est l'extension déductive de la relation sous-déductive $\rightarrow_{\mathcal{P}rg}$.

Qui plus est, on peut remplacer α par toute autre variable et conserver les relations. Ainsi, avec $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$, on a pour toute variable x la relation $\Sigma[x] \Rightarrow_{\mathcal{P}rg} \Sigma'[x]$.

Revenons maintenant à notre sémantique. Comme je l'ai dit un peu plus haut, la dérivation totale à partir d'un système initial Σ_{init} produit beaucoup trop d'équations: elle contient en fait la trace de l'exécution du programme.

Or, mon objectif dans cette thèse est de définir des transformations de programme en montrant qu'elles ne changent pas la sémantique du programme transformé. Je dois donc définir une sémantique qui mette en avant *quelles* sont les valeurs calculées par un programme, et non pas *comment* elles sont calculées.

Je continue avec l'exemple de la longueur d'une liste. A partir du système initial $\{\alpha = (cons\ 5\ (cons\ 6\ nil))\}$, les seules équations qui nous intéressent sont donc celles concernant la variable $\alpha.length$. Et même, il me faut éliminer toute équation faisant intervenir une autre variable, car elle serait une trace de l'exécution du programme. Il ne reste alors plus que l'équation $\alpha.length = 2$. C'est bien celle que je voulais obtenir.

Finalement, je peux formaliser cette attente en considérant que $\mathcal{P}rg$ établit une relation entre, d'une part, une équation de la forme $x = v$ et d'autre part, une équation de la forme $x.length = v'$, où v et v' sont des valeurs. Cependant, il faut tenir compte également des attributs paramètres qui peuvent être nécessaires à chaque attribut résultat. Pour illustrer cette situation, je reprends l'exemple de l'inversion de liste (voir section 3.3.3):

$$\left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.rev = \alpha.2.rev \\ cons & \rightarrow & \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ nil & \rightarrow & \alpha.rev = \alpha.rev\_h \end{array} \right\}$$

Pour cette exemple, la relation intéressante est celle qui relie, d'une part, deux équations $x = v_1$ et $x.rev\_h = v_2$ et d'autre part, une équation $x.rev = v_3$. Je propose donc la sémantique suivante.

3.5.5 Sémantique Équationnelle (définition)

J'ai besoin de faire une petite remarque technique concernant les ensembles et les séquences d'attributs. Si $A \subset \mathcal{A}tt$ est un ensemble d'attributs, alors en considérant un ordre arbitraire sur les noms d'attributs, on peut ordonner les éléments de A en une séquence $[a_1 \ldots a_n]$. Pour éviter toute confusion, je note cette séquence $\overline{A}$. Quand A est réduit à un unique élément, je me passerai des parenthèses et des crochets pour ne pas encombrer inutilement les notations.

Définition 3.5.5 (Sémantique Équationnelle)
Soient un ensemble d'attributs paramètres $P \subset \mathcal{P}rm$, et un ensemble d'attributs résultats $R \subset \mathcal{R}es$. Je considère la sémantique équationnelle de $\mathcal{P}rg$ par rapport *à la paire (P, R): c'est une relation entre tuples de valeurs, notée $[\![\mathcal{P}rg]\!]_{(P,R)}$ et définie par:*

$$(v, v_1 \ldots v_n)[\![\mathcal{P}rg]\!]_{(P,R)}(w_1 \ldots w_m) \quad \Leftrightarrow \quad \Sigma_{init} \Rightarrow_{\mathcal{P}rg} \Sigma_{final}$$

Avec $\overline{P} = [p_1 \ldots p_n]$, $\overline{R} = [r_1 \ldots r_m]$ et:

$$\Sigma_{init} = \left\{ \begin{array}{l} \alpha = v \\ \alpha.p_1 = v_1 \\ \ldots \\ \alpha.p_n = v_n \end{array} \right. \qquad \Sigma_{final} = \left\{ \begin{array}{l} \alpha.r_1 = w_1 \\ \ldots \\ \alpha.r_m = w_m \end{array} \right.$$

3.5.6 Dépendances approchées

Comme on l'a vu dans la sémantique d'un programme équationnel, il y a une certaine dépendance entre les attributs résultats et les attributs paramètres. Ainsi, quand $[\![\mathcal{P}rg]\!]_{(P,R)}$ n'est pas vide, on peut dire que pour calculer R, il faut P. On peut aussi dire que R dépend de P, ou que R est une fonction de P ou qu'il existe une fonction donnant R à partir de P, etc. Je noterai cette relation de dépendance par $\mathcal{D}ep(R) = P$.

Cependant, il se peut aussi que $[\![\mathcal{P}rg]\!]_{(P',R)}$ soit également non vide. Que dire alors de la dépendance de R? Dans le premier cas, on aurait aimé dire que R dépend de P, mais ici, on voudrait que R dépende de P'. Dans la suite j'aurais juste besoin de connaître un ensemble d'attributs P qui détermine la valeur des attributs R. Je choisirai donc une approximation dans laquelle on aurait pour le cas précédent $\mathcal{D}ep(R) \supset P \cup P'$. Dans le cas général, on aura:

Définition 3.5.6 (Dépendances Approchées)
Soit $\mathcal{D}ep$ une relation entre ensembles d'attributs.

$\mathcal{D}ep$ est une relation de dépendance approchée si elle vérifie:

$$\mathcal{D}ep(R) \supset \bigcup \{P \mid [\![\mathcal{P}rg]\!]_{(P,R)} \neq \emptyset\}$$

Pour calculer une relation $\mathcal{D}ep$ on est pas obligé de calculer les relations $[\![\mathcal{P}rg]\!]_{(P,R)}$, ce qui est heureux car c'est calcul onéreux. Je donne à la section 6.2 un algorithme assez simple pour calculer rapidement la relation $\mathcal{D}ep$.

3.6 Transformations

Dans cette section, je fais une présentation générale de ce qu'est une transformation de programme dans le cadre de la sémantique équationnelle. Plusieurs points sont à examiner. Mais tout d'abord, je tiens à mettre en garde le lecteur. Par rapport aux notions classiques de transformation, j'introduis une démarche originale.

Considérons un programmes $\mathcal{P}rg$ et une "transformation", c'est à dire un autre programme $\mathcal{P}rg'$. Dans un cadre classique, on se demanderait si il est bon de *remplacer* le programme $\mathcal{P}rg$ par $\mathcal{P}rg'$. Dans le cadre de la sémantique équationnelle, je me propose d'*ajouter* le programme $\mathcal{P}rg'$ à $\mathcal{P}rg$, et de considérer que le résultat de la transformation est le programme $\mathcal{P}rg \cup \mathcal{P}rg'$. Ainsi, le problème du choix peut être factorisé et vu comme une autre forme de transformation, appelée réduction de programme. J'y reviendrai dans le section 3.7 sous forme de commentaires, puis dans la suite de la thèse dans les chapitres 5 et 8.

Voici maintenant les différents points que je vais aborder. Premièrement, je vais définir grâce la sémantique équationnelle une équivalence entre programmes. Ensuite, je donnerai une définition d'une transformation. Enfin, je parlerai d'une transformation particulière qui sert de clef à tout faire pour les autres transformations: l'évaluation partielle.

3.6.1 Équivalence de Programmes

Un programme $\mathcal{P}rg$ peut être la réunion d'une multitude de différents programmes ayant chacun leurs attributs paramètres et résultats, mélangés dans n'importe que ordre. Ce sera surtout le cas quand on aura appliqué quelques transformations.

Or la sémantique équationnelle est définie par rapport à des paires (P, R). Rappelons un peu l'idée intuitive: la sémantique $[\![\mathcal{P}rg]\!]_{(P,R)}$ est une fonction qui met en relation un tuple de valeurs *initiales* avec un tuple de valeurs *finales*. Les premières sont les valeurs pour les attributs P et les secondes celle des attributs R.

Ce qui veut dire que l'on peut regarder un même programme $\mathcal{P}rg$ avec plusieurs paires (P, R) différentes. Par exemple, considérons le programme suivant:

$$\begin{array}{rl} \mathcal{P}rg = & \left\{ \begin{array}{lcl} cons & \rightarrow & \alpha.length = (+\ 1\ \alpha.2.length) \\ nil & \rightarrow & \alpha.length = 0 \end{array} \right\} \\ \cup & \left\{ \begin{array}{lcl} cons & \rightarrow & \alpha.rev = \alpha.2.rev \\ cons & \rightarrow & \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ nil & \rightarrow & \alpha.rev = \alpha.rev\_h \end{array} \right\} \end{array}$$

Deux paires (P, R) sont intéressantes: $(\emptyset, length)$ pour la longueur d'une liste et (rev_h, rev) pour son inversion. On peut d'ailleurs observer ce même programme avec la paire $(rev\_h, \{\ rev\ length\ \})$. Cela signifie simplement que les deux calculs sont observés simultanément sur la même liste.

Quand on a deux programmes $\mathcal{P}rg$ et $\mathcal{P}rg'$ que l'on veut comparer, on imagine alors assez bien qu'il faille se donner une ou plusieurs paires (P, R) communes aux deux programmes. Je définis ainsi la notion de champs d'observation:

Définition 3.6.1 (Champs d'observation)
Un champs d'observation $\mathcal{O}bs$ est un ensemble de paires (P, R), où P est un ensemble d'attributs paramètres et R un ensemble d'attributs résultats.

Deux programmes seront alors équivalents pour un champs d'observation $\mathcal{O}bs$ quand leurs sémantiques équationnelles, par rapport à toute paire $(P, R) \in \mathcal{O}bs$ sont égales. Je note $\sim_{\mathcal{O}bs}$ cette relation d'équivalence et on a:

Définition 3.6.2 (Équivalence)
Étant donnés deux programmes $\mathcal{P}rg$ et $\mathcal{P}rg'$ et un champs d'observation $\mathcal{O}bs$, on a:

$$\mathcal{P}rg \sim_{\mathcal{O}bs} \mathcal{P}rg' \Leftrightarrow (\forall (P, R) \in \mathcal{O}bs)\ [\![\mathcal{P}rg]\!]_{(P,R)} = [\![\mathcal{P}rg']\!]_{(P,R)}$$

Je voudrais enfin introduire une notation qui me sera bien utile dans les chapitres suivants, et qui est liée à la notion de champs d'observation.

Définition 3.6.3 (Systèmes Initiaux et Finaux)
Soit $\mathcal{O}bs$ un champs d'observation. L'ensemble des systèmes initiaux du champs $\mathcal{O}bs$ est défini par:

$$\Sigma \in \mathcal{I}nit(\mathcal{O}bs) \Leftrightarrow \Sigma = \left\{ \begin{array}{ccc} \alpha & = & v \\ \alpha.p_1 & = & v_1 \\ \ldots & & \\ \alpha.p_n & = & v_n \end{array} \right\}$$

où v, $v_1 \dots v_n$ sont des valeurs et il existe une paire $(P, R) \in \mathcal{O}bs$ telle que $\overline{P} = [p_1 \dots p_n]$.

De même, les systèmes finaux du champs $\mathcal{O}bs$ sont définis par:

$$\Sigma \in \mathcal{F}inal(\mathcal{O}bs) \Leftrightarrow \Sigma = \left\{ \begin{array}{ccc} \alpha.r_1 & = & w_1 \\ \dots & & \\ \alpha.r_m & = & w_m \end{array} \right\}$$

où $w_1 \dots w_m$ sont des valeurs et il existe une paire $(P, R) \in \mathcal{O}bs$ telle que $\overline{R} = [r_1 \dots r_m]$.

3.6.2 Transformation

Une transformation $\mathcal{T}rans$ est une application associant à un programme origine $\mathcal{P}rg$ un programme $\mathcal{P}rg'$. Je dirai qu'une transformation $\mathcal{T}rans$ est *correcte* pour le champs d'observation $\mathcal{O}bs$ quand $\mathcal{P}rg$ et $\mathcal{P}rg'$ sont équivalents dans ce champs.

Définition 3.6.4 (Transformation correcte)
La transformation $\mathcal{T}rans$ est correcte pour le programme $\mathcal{P}rg$ et le champs d'observation $\mathcal{O}bs$ si et seulement si on a:

$$\mathcal{T}rans(\mathcal{P}rg) \sim_{\mathcal{O}bs} \mathcal{P}rg$$

Évidemment, cette définition n'est jamais utilisée directement. En général, on utilise plutôt des critères qui sont suffisant pour prouver qu'une transformation est correcte. Et puis surtout, en pratique, la fonction $\mathcal{T}rans$ sera toujours construite sur un même schéma. Avant de détailler ce point, je voudrais définir une transformation particulière, l'évaluation partielle, car elle sert d'outil pour la plupart des autres transformations.

3.6.3 Évaluation Partielle

L'idée est la suivante. Supposons que l'on ait :

$$\{\alpha = (c\ \alpha.1 \dots \alpha.n)\} \Rightarrow_{\mathcal{P}rg} \Sigma$$

Alors, par universalité de la relation $\Rightarrow_{\mathcal{P}rg}$, on peu immédiatement en déduire que pour tout x on a:

$$\{x = (c\ x.1 \dots x.n)\} \Rightarrow_{\mathcal{P}rg} \Sigma[x]$$

Or, en utilisant judicieusement les règles de dérivation pour les sous-termes et les substitutions, il est équivalent de dire:

$$(\forall x)(\forall t_1 \dots t_n) \quad \{x = (c\ t_1 \dots t_n)\} \Rightarrow_{\mathcal{P}rg} \Sigma[x]$$

Cela signifie que l'on peut introduire un raccourci en ajoutant au programme les équations $c \rightarrow y = t$ pour $y = t \in \Sigma$ afin qu'elles soient produites directement sans avoir besoin de dérouler toute la dérivation à chaque fois. En d'autres termes, la transformation suivante est correcte:

Définition 3.6.5 (Évaluation Partielle)
La transformation appelée évaluation partielle, notée EP, est définie ainsi:

$$EP(\mathcal{P}rg) = \bigcup_{c \in \mathcal{C}ons} \{c \to y = t \mid y = t \in \Sigma_c\}$$

le système Σ_c étant choisit pour tout c en respectant la relation:

$$\{\alpha = (c\ \alpha.1 \ldots \alpha.n)\} \Rightarrow_{\mathcal{P}rg} \Sigma_c$$

Je reparlerai plus en détails de certaines particularités de cette transformation au chapitre 5, mais l'essentiel vient d'en être dit. Je signale également que calculer les systèmes Σ_c ci-dessus en un temps raisonnable est un problème technique délicat et que le chapitre 7 s'attarde longuement sur ce point.

3.7 Commentaires et Perspectives

Je voudrais faire ici quelques remarques sur les possibilités qu'offrent ce formalisme de sémantique équationnelle. Pour cela, je vais m'appuyer sur l'exemple suivant:

$$\mathcal{P}rg = \left\{ \begin{array}{ccl} main & \to & \alpha.r = (+\ \alpha.L_1.r\ \alpha.p) \\ " & \to & \alpha.L_1 = (fact) \\ " & \to & \alpha.L_1.p = 5 \\ fact & \to & \alpha.r = \alpha.L_1.f \\ " & \to & \alpha.L_1 = (<\ 1\ \alpha.p) \\ " & \to & \alpha.L_1.f\_x = \alpha.p \\ true & \to & \alpha.f = (*\ \alpha.f\_x\ \alpha.L_2.r) \\ " & \to & \alpha.L_2 = (fact) \\ " & \to & \alpha.L_2.p = (-\ \alpha.f\_x\ 1) \\ false & \to & \alpha.f = 1 \end{array} \right\}$$

On reconnaît dans ce programme celui de la section 3.3.4 qui calcule la fonction factorielle. J'y ai juste ajouté le constructeur *main* qui permet de définir une fonction qui ajoute à son argument la factorielle de 5 (avec $5! = 120$). Un équivalent de ce programme en langage fonctionnel serait :

```
let fact x = f x (< 1 x)

let f x = fun
| true  -> (* x (fact (- x 1)))
| flase -> 1

let main x = (+ (fact 5) x)
```

Revenons maintenant au programme équationnel. Par exemple, on aura:

$$\left\{\begin{array}{lcl}\alpha & = & (main)\\ \alpha.p & = & 7\end{array}\right\} \Rightarrow_{\mathcal{P}rg} \{\ \alpha.r \quad = \quad 127\ \}$$

Soit alors le champs d'observation défini par $\mathcal{O}bs = \{\ (p,r)\ \}$. La sémantique de $\mathcal{P}rg$ dans ce champs est facile à exprimer. En effet, on a $(v, v_p)[\![\mathcal{P}rg]\!]_{(p,r)}(v_r)$ si et seulement si:

- soit $v = (fact)$, et $v_r = v_p!$,
- soit $v = (main)$, et $v_r = (+\ 5!\ v_p)$.

Dans le cas où $v = (main)$, on peut aussi dire que $v_r = (+\ 120\ v_p)$, avec $5! = 120$. Or, sur cet exemple, l'évaluation partielle donne justement un nouveau programme dont voici un extrait:

$$\mathcal{P}rg' = \left\{\begin{array}{lcl} main & \rightarrow & \alpha.r = (+\ \alpha.L_1.r\ \alpha.p)\\ \text{''} & \rightarrow & \alpha.r = (+\ 120\ \alpha.p)\\ \text{''} & \rightarrow & \alpha.L_1 = (fact)\\ \text{''} & \rightarrow & \alpha.L_1.p = 5\\ (\ldots) & & \end{array}\right\}$$

Je signale par ailleurs que l'on a $\mathcal{P}rg \sim_{\mathcal{O}bs} \mathcal{P}rg'$. On retrouve donc ici la caractéristique d'une transformation en sémantique équationnelle: les transformations se font par augmentation de programme. On observe alors que dans cet exemple il y maintenant deux équations qui définissent la variable $\alpha.r$ (dans le cas du constructeur *main*).

Déterminisme: je considère que le programme $\mathcal{P}rg$ est *déterministe* parce que pour chaque constructeur c et chaque variable x, il y a au plus une équation de programme de la forme $c \rightarrow x = t$ dans $\mathcal{P}rg$.

Traduction: les programmes équationnels déterministes sont intéressants car ils peuvent facilement être traduits en des programmes fonctionnels. On peut même les traduire dans bien d'autres formalismes. Ce type de traduction est développé dans le chapitre 9 et permet en pratique d'utiliser les transformations de programmes équationnels pour des programmes fonctionnels.

Non-déterminisme: je considère que le programme $\mathcal{P}rg'$ est *non-déterministe* parce que pour certains constructeurs c et certaines variables x, il y a plus d'une équation de programme de la forme $c \rightarrow x = t$. Ce genre de programme n'est pas directement traduisible dans un langage fonctionnel car il y a en certains endroits plusieurs possibilités pour résoudre le système initial.

Stratégies: dans le cas de $\mathcal{P}rg'$, on a deux façons de calculer l'attribut r pour le constructeur *main*. Le fait que l'évaluation partielle soit une transformation correcte prouve que l'on peut arbitrairement choisir l'une ou l'autre voie. J'en viens donc à la notion de *stratégie* de résolution d'un système. Soient par exemple les deux systèmes suivants:

$$\Sigma_{init} = \left\{ \begin{array}{lcl} \alpha & = & (main) \\ \alpha.p & = & 7 \end{array} \right\} \quad \Sigma_{final} = \left\{ \begin{array}{lcl} \alpha.r & = & 127 \end{array} \right\}$$

Je vais montrer qu'il existe deux stratégies de calcul pour obtenir la relation $\Sigma_{init} \Rightarrow_{\mathcal{P}rg'} \Sigma_{final}$.

Stratégie 1: On commence par appliquer les équations de $\mathcal{P}rg'$ qui définissent les variables $\alpha.L_1$ et $\alpha.L_1.p$, on obtient:

$$\Sigma_{init} \quad \Rightarrow_{\mathcal{P}rg'} \quad \left\{ \begin{array}{lcl} \alpha.L_1 & = & (fact) \\ \alpha.L_1.p & = & 5 \end{array} \right\}$$

Ensuite, on déroule une dérivation qui calcule $\alpha.L_1.r$, ce qui donne:

$$\left\{ \begin{array}{lcl} \alpha.L_1 & = & (fact) \\ \alpha.L_1.p & = & 5 \end{array} \right\} \quad \Rightarrow_{\mathcal{P}rg'} \quad \left\{ \begin{array}{lcl} \alpha.L_1.r & = & 120 \end{array} \right\}$$

Parallèlement, on applique la première équation qui définit la variable $\alpha.r$ pour le constructeur *main*, on trouve:

$$\Sigma_{init} \quad \Rightarrow_{\mathcal{P}rg'} \quad \{\alpha.r = (+\ \alpha.L_1.r\ \alpha.p)\}$$

Enfin, par substitution et transitivité, on trouve:

$$\Sigma_{init} \quad \Rightarrow_{\mathcal{P}rg'} \quad \{\alpha.r = (+\ 120\ 7)\} \quad \Rightarrow_{\mathcal{P}rg'} \quad \Sigma_{final}$$

Remarquons d'ailleurs que cette stratégie est l'unique résolution qui fonctionne si on se limite au programme $\mathcal{P}rg$ (et non en utilisant $\mathcal{P}rg'$). Mais on peut aller beaucoup plus vite, en remarquant que l'on peut utiliser une autre stratégie.

Stratégie 2: On commence cette fois par appliquer la deuxième équation qui définit la variable $\alpha.r$ pour le constructeur *main*, on trouve alors directement:

$$\Sigma_{init} \quad \Rightarrow_{\mathcal{P}rg'} \quad \{\alpha.r = (+\ 120\ \alpha.p)\}$$

Puis, par substitution, on trouve:

$$\Sigma_{init} \quad \Rightarrow_{\mathcal{P}rg'} \quad \{\alpha.r = (+\ 120\ 7)\} \quad \Rightarrow_{\mathcal{P}rg'} \quad \Sigma_{final}$$

Le lecteur pourra remarquer que dans chacune des stratégies, l'ordre dans lequel les équations sont évaluées est figé et que la résolution est dans un certain sens déterministe.

Sémantique Opérationnelle: l'existence des stratégies et leur calcul est un point auquel qui je consacre entièrement le chapitre 8. Ce qui m'intéresse ici est le fait que, si l'on choisit une stratégie, alors on peut en extraire un programme déterministe. Dans mon exemple, on obtient deux programmes $\mathcal{P}rg_1$ et $\mathcal{P}rg_2$ qui sont tout simplement:

$$\begin{array}{l} \mathcal{P}rg_1 = \mathcal{P}rg \\ \mathcal{P}rg_2 = \{ \ \ main \ \ \rightarrow \ \ \alpha.r = (+\ 120\ \alpha.p) \ \ \} \end{array}$$

On a clairement $\mathcal{P}rg \sim_{\mathcal{O}bs} \mathcal{P}rg_1$ et $\mathcal{P}rg \sim_{\mathcal{O}bs} \mathcal{P}rg_2$. On peut donc se servir de cette notion de stratégie pour définir une transformation un peu particulière, qui réduit un programme au lieu de l'augmenter. Évidemment, il faudra être capable de choisir entre $\mathcal{P}rg_1$ et $\mathcal{P}rg_2$, point que je développe à la section 5.8.

Par ailleurs, les deux programmes $\mathcal{P}rg_1$ et $\mathcal{P}rg_2$ sont déterministes. On pourra donc les traduire en deux programmes fonctionnels différents. Il est dans ce cas facile de s'assurer que le deuxième est bien plus efficace que le second, et on aura ainsi obtenu une transformation intéressante du programme initial.

3.8 Notice de lecture

Je voudrais terminer ce chapitre par une description un peu plus précise des chapitres suivants. Ils sont relativement indépendants les uns des autres et on peut donc les aborder dans un ordre quelconque.

- Le chapitre 4 donne quelques résultats obtenus en appliquant des transformations équationnelles à des programmes fonctionnels.
- Le chapitre 5 est un exposé des transformations de programme. On y trouvera pas les preuves de correction mais des exemples et des figures expliquant intuitivement pourquoi ces transformations sont correctes. Il se termine par des indications techniques permettant d'implémenter efficacement ces transformations.
- Le chapitre 6 contient les preuves des transformations. Plus exactement, on y trouvera les preuves des théorèmes importants. C'est un chapitre difficile et technique, avec quelques belles démonstrations.
- Le chapitre 7 explique comment implémenter un opérateur d'évaluation partielle (une généralisation de l'opérateur EP définie à la section 5.1) de manière efficace. C'est un chapitre très important sur le plan pratique car ces opérateurs d'évaluation partielle jouent un rôle crucial dans les transformations de programme. Par ailleurs, le problème à résoudre n'est pas si simple et le chapitre explique comment diminuer fortement sa complexité jusqu'à un niveau raisonnable.
- Le chapitre 8 traite des stratégies et de leur calcul. C'est en fait une adaptation des travaux sur les grammaires attribuées de la classe `Fnc` (voir par exemple [26]). Le problème, tout d'abord résolu sur le plan théorique, est ensuite raffiné pour être calculable de manière efficace. On y trouvera

donc des indications d'implémentation qui m'ont permis de réaliser un prototype.

- Le chapitre 9 plonge les résultats de la sémantique équationnelles dans la programmation fonctionnelle, en décrivant deux traductions, une pour chaque sens (fonctionnel vers équationnel et réciproquement). Pour la traduction d'un programme fonctionnel, même d'ordre supérieur, la traduction est très simple. En revanche, l'inverse est une opération plus délicate si l'on attend un résultat lisible. Je n'en donne donc qu'une version simple, tout en proposant quelques raffinements possibles.
- Le chapitre 10 contient quelques définitions à propos des relations, des graphes, etc. Quelques notions algébriques originales ou amusantes dont j'ai eu besoin au cours des précédents chapitres y sont exposées plus en détails.

Enfin, je rappelle à toute fin utile qu'un index des définitions et des théorèmes permet de retrouver la ou les pages correspondantes, en cas de besoin.

Chapitre 4

Résultats

Dans ce chapitre, je présente quelques résultats pratique obtenus par des transformations de programmes en sémantique équationnelle. Bien entendu, pour des problèmes de lisibilité, je ne donnerai pas les détails des transformations effectuées. D'autant que leur définition n'est donnée qu'au chapitre 5.

Donc, le lecteur trouvera dans les sections qui suivent des exemples sous forme de programmes fonctionnels. Je ne donnerai pas toutes les étapes intermédiaires des traductions en programme équationnels ni leurs transformations successives. Cependant, j'essaierai dans la mesure du possible de donner des étapes intermédiaires des processus de transformations, mais uniquement sous forme de programmes fonctionnels.

D'un point de vue pratique, ces résultats ont été obtenus par un prototype d'implémentation des algorithmes décris dans les chapitres 5, 7, 8 et 9. Ce prototype est complètement automatique et a été écris en Objective Caml[1]. Une présentation rapide de ce prototype est donnée à la section 4.4

4.1 Déforestation dans les accumulateurs

Je reprends un exemple de la section 2.4 pour examiner le comportement des méthodes de déforestations classiques:

```
let append xs ys = match xs with
| cons a b -> cons a (append b ys)
| nil -> ys

let main x y z = append (append x y) z
```

Avec une méthode de déforestation fonctionnelle (algorithme de Wadler modifié pour éviter les cas de non terminaison, ou bien fusion d'hylomorphismes

1. Langage de programmation fonctionnelle développé par l'INRIA-Rocquencourt, Cf. http://pauillac.inria.fr/ocaml/index.html.

par exemple), on peut obtenir le programme partiellement déforesté suivant:

```
let app2 xs ys zs = match xs with
| cons a b -> cons a (app2 b ys zs)
| nil -> append ys zs

let main x y z = app2 x y z
```

La déforestation est incomplète car le calcul de la liste intermédiaire est toujours présent, on a juste déplacé l'un des appels à la fonction `append` au niveau d'un constructeur `nil`.

En sémantique équationnelle, on trouve immédiatement une première composition qui donne ce premier résultat. Ensuite, la sémantique équationnelle permet d'introduire de nouvelles équations dans le programme, à la différence des méthodes fonctionnelles classiques. Mais hélas, ces nouvelles équations ne peuvent pas être représentées par des fonctions. Je ne peux donc pas illustrer ici cette étape de transformation. En revanche, après évaluation partielle, on peux extraire du programme équationnel transformé le programme fonctionnel suivant:

```
let app3 xs ws = match xs with
| cons a b -> cons a (app3 b ws)
| nil -> ws

let app4 xs zs = match xs with
| cons a b -> app4 b zs
| nil -> zs

let main x y z = (app3 x (append y (app4 x z))
```

On découvre alors une propriété intéressante: la fonction `app4` renvoie toujours son deuxième argument. Au niveau équationnel, cette propriété est très facilement et automatiquement détectée et la transformation associée donne le programme suivant :

```
let main x y z = (app3 x (append y z))
```

Et on remarquera également que la fonction `app3` est en tout point identique à la fonction `append`. On peut donc voir que les méthodes équationnelles donnent la déforestation attendue.

Un exemple mixte: dans cette section, je voudrais présenter un exemple de déforestation dans un accumulateur que l'on peut entièrement interpréter au niveau fonctionnel. C'est donc un enrichissement de l'algorithme initial de

Wadler que l'on retrouve par ailleurs dans ??. Voici le programme initial :

```
let length = fun
| cons a b -> (+ 1 (length b))
| nil -> 0

let rev p = fun
| cons a b -> rev (cons a p) b
| nil -> p

let main x = length (rev x nil)
```

L'astuce ici consiste à introduire une fonction `g` définie par :

```
forall(p,x):   g p (length p) x = length (rev p x)
```

On peut alors dire que, dans la définition de `g`, le deuxième argument est toujours égal à la longueur du premier. Ce qui permet de dériver le programme suivant :

```
let g p lp x = match x with
| cons a b -> length (rev (cons a p) b)
| nil -> length p

let main x = g nil (length nil) x
```

Utilisant alors le fait que `length p=lp` on peut déduire après évaluation partielle :

```
let g p lp x = match x with
| cons a b -> g (cons a p) (+ 1 lp) b
| nil -> lp

let main x = g nil 0 x
```

On s'aperçoit ici que le paramètre `p` de la fonction `g` n'est jamais utilisé, donc on arrive finalement à :

```
let g2 lp x = match x with
| cons a b -> g2 (+ 1 lp) b
| nil -> lp

let main x = g2 0 x
```

La déforestation est réussie !

Autre exemple: je reviens maintenant sur l'exemple de l'introduction, pour en donner une autre description. Je rappelle le programme à transformer :

```
let flat p = fun
| node t1 t2 -> flat (flat p t2) t1
| leaf n -> (cons n p)

let main t = rev nil (flat nil t)
```

Le but est ici d'éviter de construire la liste intermédiaire construite par la fonction `flat` avant qu'elle ne soit inversée par `rev`. Cette transformation est délicate car elle nécessite de modifier le parcours effectué par la fonction `flat`, et plus généralement, c'est un problème de déforestation sur les accumulateurs.

Voyons un peu comment les transformations équationnelles se sortent de ces mauvais pas. Tout d'abord, une déforestation très simple et classique donne le programme suivant :

```
let f1 p' p = fun
| node t1 t2 -> f1 p' (flat p t2) t1
| leaf n -> rev (cons n p') p

let main t = (f1 nil nil t)
```

C'est ici que les problèmes se posent. Les transformations équationnelles génèrent à cette étape un programme équationnel qui ne peut pas se traduire en d'autres fonctions que les précédentes. Ce n'est qu'après évaluation partielle que l'on peut obtenir un premier résultat, qui est composé de deux fonctions dont on ne comprend pas bien le rôle au premier regard[2]. Ces deux fonctions sont :

```
let f2 p = fun
| node t1 t2 -> f2 (f2 p t2) t1
| leaf n -> p

let f3 p = fun
| node t1 t2 -> f3 (f3 p t1) t2
| leaf n -> (cons n p)

let main t = (f2 (f3 nil t) t)
```

C'est alors qu'on se rend compte que `f2` est simplement la fonction identité, et que l'on obtient finalement par une dernière transformation équationnelle le résultat suivant :

```
let main t = (f3 nil t)
```

2. Je rappelle qu'en sémantique équationnelle, on peut avoir des équations qui ne se traduisent pas toujours en des fonctions. Voir le chapitre 8 et la section 5.5.

Dernier exemple: un autre résultat exactement similaire est obtenu en composant `rev` avec elle-même :

```
let main l = rev nil (rev nil l)
```

Ce programme est transformé en :

```
let f4 p = fun
| cons a b -> cons a (f4 p b)
| nil -> p

let main l = f4 nil l
```

Suite à une spécialisation de la fonction `f4` on trouve alors :

```
let f5 = fun
| cons a b -> cons a (f5 b)
| nil -> nil

let main l = f5 l
```

C'est déjà un beau résultat, considérant qu'aucune méthode de déforestation classique ne réussit à prouver que la fonction `rev` composée avec elle-même peut se réduire à une simple duplication de liste. En fait, certaines méthodes y arrive, mais à l'aide d'une fonction intermédiaire qu'il faut introduire manuellement, alors que la sémantique équationnelle l'obtient directement.

Finalement, je pourrais même définir une transformation qui repère les fonctions de duplications (ce serait une variante de la section 5.6) et on aurait alors :

```
let main l = l
```

4.2 Déforestation de programmes non linéaires

Je reprends ici l'exemple de l'introduction du chapitre 3, dont je rappelle le programme fonctionnel ici:

```
let exptree = fun
  | succ n -> let t = exptree n in (node t t)
  | zero -> (leaf)

let size h = fun
  | (node a b) -> size (size h a) b
  | (leaf) -> (succ h)

let main n = size zero (exptree n)
```

C'est une déforestation classique qui fonctionne aussi bien en sémantique équationnelle qu'en programmation fonctionnelle, contrairement au cas des grammaire attribuées classiques. On obtient bien le programme suivant:

```
let exp h = fun
  | succ n -> exp (exp h a) b
  | zero -> (succ h)

let main n = exp zero n
```

4.3 Déforestation des continuations

En, voici dernier exemple qui montre comment l'ordre supérieur, en particulier l'utilisation des continuations, peut être sujet à déforestation. Voici le programme initial, inspiré de Bird et Merteens []:

```
let repmin = fun
  | node t1 t2 ->
     let (m1,k1) = repmin t1 in
     let (m2,k2) = repmin t2 in
     ( (min m1 m2) , (fun m -> node (k1 m) (k2 m)) )
  | leaf n ->
     ( n , (fun m -> leaf m) )

let main t =
  let (m,k) = repmin t in (k m)
```

Ce programme est une jolie version avec ordre supérieur d'un algorithme qui remplace les feuilles d'un arbre binaire par la plus petite d'entre elles. Le programme est compliqué, l'expression `(repmin t)` renvoyant un couple `(m,k)`, où `m` est la feuille minimum de l'arbre `t` et où `k` est une fonction qui reconstruit le même arbre avec une même valeur aux feuilles.

En sémantique équationnelle, ce programme est transformé en une forme où il n'y a pas de cas particulier dû à l'ordre supérieur. A la différence des méthodes fonctionnelles classiques qui doivent être modifiées pour tenir compte de l'ordre supérieur, ici la composition *classique* des programmes *équationnels* donne immédiatement le résultat :

```
let f1 = fun
  | node t1 t2 -> (min (f1 t1) (f1 t2))
  | leaf n -> n

let f2 m = fun
  | node t1 t2 -> (node (f2 m t1) (f2 m t2))
  | leaf n -> leaf m

let main t = f2 (f1 t) t
```

Il y bien eu déforestation, mais ce n'est pas si facile à voir. La structure qui a été déforestée est la continuation `fun m ->...` qui est construite dans la fonction `repmin` et qui alourdit l'exécution du programme. Des mesures sur l'exécution des deux programmes montrent que la version non déforestée consomme 50 pour cent de mémoire de plus que la version déforestée.

4.4 Implémentation

Dans cette section, je présente rapidement la structure complète d'un prototype du système de transformation de programme fonctionnels par sémantique équationnelle.

4.4.1 Langage Fonctionnel

Le langage fonctionnel considéré est exactement celui du chapitre 2, mais augmenté d'un certain nombre de constructions syntaxiques bien utiles. Ainsi, on trouve dans ce langage:

- des constructions conditionnelles `if`... `then`... `else`.
- des pattern-matching emboîtés: `match`... `with`...
- des tuples: (`exp`,... ,`exp`)
- des construction fonctionnelles explicites: `fun x ->exp`.
- des déclarations locales: `let x,...,x = exp`.

Toutes ces constructions sont faciles à coder dans le langage fonctionnel initial.

4.4.2 Chaîne de Transformation

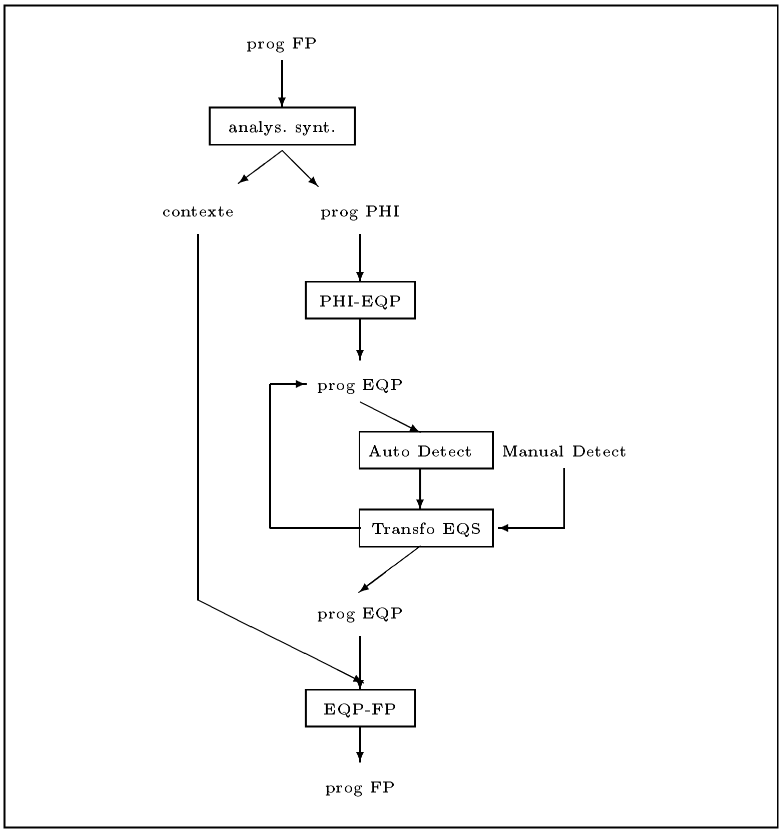
Pour faire des transformations de programmes, un certain nombre d'étapes se succèdent. La procédure est totalement automatique, bien que l'utilisateur puisse intervenir en paramétrant certaines opérations.

Dans un premier temps, une analyse syntaxique est effectuée pour convertir le programme fonctionnel en un programme `PHI` qui respecte strictement le langage du chapitre 2. Parallèlement, cette analyse donne aussi des informations sur le contexte du programme, comme par exemple les noms des fonctions, etc.

Dans un second temps, ce programme `PHI` est transformé en un programme équationnel `EQP`. Ensuite, ce programme va être transformé. Les paramètres des transformations sont, soit automatiquement déterminés par un programme spécial (AutoDetect), soit spécifiées par l'utilisateur (ManualDetect). Éventuellement, plusieurs cycles peuvent se succéder. Le nombre de cycles ou les critères d'arrêts sont paramétrables dans le programme (AutoDetect).

Une fois le programme équationnel transformé, il ne reste plus qu'à le transformer en un nouveau programme fonctionnel. On se sert ici du contexte syntaxique généré dans le première étape pour récupérer les noms de fonctions initiaux et rendre le programme final lisible.

Cette procédure est illustrée dans la figure ci-dessous:



Chapitre 5

Transformations de Programmes

Les transformations de programmes qui sont exposées ici s'appliquent, bien entendu, à des programmes équationnels. Cependant, elles correspondent parfois à des transformations de programmes fonctionnels. On trouvera donc deux sources d'inspiration. L'une provient de la vision fonctionnelle, l'autre est purement équationnelle. Parfois, faire le lien entre les deux ne sera pas simple, et je ne m'y aventurerai pas trop en détails.

Ce chapitre ne contient pas les preuves de correction mais s'appuie sur des idées générales. On aura ainsi, je l'espère, un éventail des techniques utilisables pour définir d'autres transformations de programmes.

Les transformations présentées ici sont toutes correctes, sauf une qui ne l'est que pour certaines formes de dérivations (Cf. section 5.8). En effet, certaines preuves nécessitent de considérer des dérivations saturées (Cf. chapitre 8). En deux mots, ce sont des dérivations qui correspondent à des algorithmes de résolutions et qui peuvent être traduits en un programme fonctionnel (Cf. chapitre 9). Ces transformations ne sont donc applicables que dans le cadre où la traduction des programmes équationnels en programmes fonctionnels sera utilisée en dernier lieu.

5.1 Évaluation Partielle

J'ai déjà parlé de l'évaluation partielle dans la section 3.6, mais je la reprends ici un peu plus en détails, avec en prime une petite extension. La transformation EP que j'avais défini à la section 3.6 permettait d'ajouter de nouvelles équations $c \to y = t$ au programme $\mathcal{P}rg$ pourvu que l'on ait:

$$\{\alpha = (c\ \alpha.1 \ldots \alpha.n)\} \Rightarrow_{\mathcal{P}rg} \{y = t\}$$

En pratique, la recherche des équations $y = t$ peut être effectuée de bien des manières différentes. Donc, à priori, il n'y a pas une mais plusieurs évaluation partielles. Pour mettre en œuvre les transformations de programmes et pour faciliter leur preuves de correction, il est utile d'introduire une nouvelle notion: celle d'opérateur d'évaluation partielle.

Définition 5.1.1 (Opérateur d'évaluation Partielle)

Une fonction OEP entre systèmes est un opérateur d'évaluation partielle pour le champs $\mathcal{O}bs$ si et seulement si, pour tout système Σ, et pour tout système final $\Sigma' \in \mathcal{F}inal(\mathcal{O}bs)$ on a:

$$\Sigma \cup OEP(\Sigma) \Rightarrow_{\mathcal{P}rg} \Sigma' \quad \Rightarrow \quad \Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$$

On impose aussi à la fonction OEP de vérifier les propriétés de monotonie suivantes:

- $OEP(\Sigma[x]) = OEP(\Sigma)[x]$
- $OEP(\Sigma_1 \cup \Sigma_2) = OEP(\Sigma_1) \cup OEP(\Sigma_2)$

La propriété essentielle d'un opérateur dévaluation partielle est qu'il ne peut pas introduire d'équation qui perturbe ou modifie les résultats observés dans le champs $\mathcal{O}bs$. Les deux propriétés de monotonies sont en général très faciles à vérifier, je n'en parlerai donc pratiquement jamais.

Exemple: une fonction f telle que $\Sigma \Rightarrow_{\mathcal{P}rg} f(\Sigma)$ est un opérateur d'évaluation partielle. En effet, $\Sigma \Rightarrow_{\mathcal{P}rg} f(\Sigma)$, donc on a aussi $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma \cup f(\Sigma)$. Donc si $\Sigma \cup f(\Sigma) \Rightarrow_{\mathcal{P}rg} \Sigma'$, on a par transitivité $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$.

Avant d'aller plus loin, je voudrais signaler que les opérateurs sont très nombreux, qu'ils sont stables par union et fermeture transitive, etc. Parmi les principaux, on peut dores et déjà mentionner les relations $\rightarrow_{\mathcal{P}rg}$ et $\Rightarrow_{\mathcal{P}rg}$.

Définition 5.1.2 (Évaluation Partielle)

Soit (EP) la relation entre systèmes telle que $\Sigma_1(EP)\Sigma_2$ si, et seulement si, il existe un opérateur d'évaluation partielle OEP tel que $\Sigma_2 \subset \Sigma_1 \cup OEP(\Sigma_1)$.

Alors, la transformation d'évaluation partielle est définie par:

$$\mathcal{T}rans_{(EP)}(\mathcal{P}rg) = \mathcal{P}rg \cup \{c \rightarrow x = t \mid \{\alpha = (c\ \alpha.1 \ldots \alpha.n)\}(EP)^* x = t\}$$

Avec ces définitions, on a le théorème suivant:

Théorème 5.1.1 (Correction de l'évaluation partielle)

$$\mathcal{T}rans_{(EP)}(\mathcal{P}rg) \sim_{\mathcal{O}bs} \mathcal{P}rg$$

La relation (EP) peut sembler un peu étrange, mais correspond seulement à l'idée que l'on peut cumuler les effets de plusieurs opérateurs d'évaluation partielle. De fait, $(EP)^*$ définit un nouvel opérateur d'évaluation partielle. De cette manière, on peut effectivement enrichir l'évaluation partielle en y incorporant de nouveaux opérateurs. Le chapitre 7 donne une implémentation efficace d'un opérateur d'évaluation partielle très puissant.

Ces opérateurs sont la base des transformations que je vais définir. Une transformation sera tout simplement l'application de l'évaluation partielle enrichie par des nouveaux opérateurs. On peut donc considérer que toutes les transformations sont effectuées *par* évaluation partielle.

Pour finir, la définition des opérateurs d'évaluation partielle garantie la correction de la transformation $\mathcal{T}rans$. Je signale également qu'en fin de chapitre, la transformation de réduction d'un programme n'est pas fondée sur l'évaluation partielle, et qu'elle n'est pas correcte pour toute forme de dérivation.

5.2 Transformations plus complexes

L'évaluation partielle limitée à l'opérateur $\Rightarrow_{\mathcal{P}rg}$ ne fait rien d'autre qu'une sorte de fermeture sur les équations d'un programme. Des transformations plus intéressantes et plus complexes vont maintenant être obtenues par d'autres opérateur d'évaluation partielle. Pour les découvrir et en construire à volonté, on peut en fait s'appuyer sur (ou s'inspirer de) quelques principes élémentaires. Je vais les décrire à l'occasion d'une transformation assez intuitive et très classique: la spécialisation.

Avant d'aborder cette transformation et les suivantes, je voudrais cependant donner quelques notations bien utiles.

Définition 5.2.1 (Constructeurs définissant R)
Soit $R \subset \mathcal{R}es$ un ensemble d'attributs résultats. Je note τ_R l'ensemble des constructeurs pour lesquels tous les attributs de R sont définis par au moins une équation:

$$\tau_R = \{c \in \mathcal{C}ons \mid (\forall r \in R)(\exists t) \quad c \to \alpha.r = t \in \mathcal{P}rg\}$$

J'ai également besoin de définir la notion d'égalité de deux variables. Intuitivement, on aurait envie de dire que x et y sont égales quand, pour toute équation $x = t$ on a une équation $y = t$ et réciproquement. En fait, on ne se pose cette question que dans le contexte d'un certain système Σ. On voudrait donc avoir la propriété $x = t \in \Sigma$ si et seulement si $y = t \in \Sigma$. On peut généraliser un peu plus, et avec l'aide des règles de substitution je propose la définition suivante:

Définition 5.2.2 (Égalité de Variables)
L'égalité, dans le contexte du système Σ, des variables x et y est notée $x =_\Sigma y$ et elle est définie par:

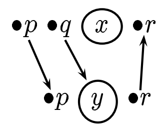
$$x =_\Sigma y \quad \Leftrightarrow \quad \Sigma \Rightarrow_{\mathcal{P}rg} \{x = y\}$$

Bien évidemment, on a la propriété voulue, à savoir que l'on a $\Sigma \Rightarrow_{\mathcal{P}rg} x = t$ si et seulement si $\Sigma \Rightarrow_{\mathcal{P}rg} y = t$. On utilisera cette notion d'égalité, car on verra que si $x =_\Sigma y$, alors la fonction définie par $f(\Sigma) = \{x = y\}$ est un opérateur d'évaluation partielle.

Figures: pour agrémenter les explications des transformations, je donne au fil du texte des petites figures sensées illustrer mon propos. Leur but précis est de montrer sous forme de diagrammes quelles sont les variables qui sont égales entre elles et quelles sont les équations dont on dispose au cours d'une dérivation. Prenons par exemple le petit système suivant (x et y sont deux variables, $r \in \mathcal{R}es$ et $p, q \in \mathcal{P}rm$ sont des attributs):

$$\Sigma = \left\{ \begin{array}{ccc} x.r & = & y.r \\ y.p & = & x.p \\ y & = & x.q \end{array} \right\}$$

Je représente ce système par le petit diagramme suivant:



Ainsi, la variable x est représentée par x, et la variable $x.r$ est simplement représentée par un $\bullet r$ à coté du x, comme dans $x \ \bullet r$. Par convention, les attributs résultats seront placés à droite et les attributs paramètres à gauche. Une flèche, comme dans $x \rightarrow y$ représente alors une équation $y = x$ (attention au sens!). Quand le système Σ permet de vérifier $x =_\Sigma y$, je le signalerai par une double flèche, comme dans $x \Longleftrightarrow y$.

Variables hors-champs: cette notion est une des clefs pour comprendre mes transformations. Si je prends deux dérivations, par exemple $\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma'_1$ et $\Sigma_2 \Rightarrow_{\mathcal{P}rg} \Sigma'_2$, alors on a naturellement $\Sigma_1 \cup \Sigma_2 \Rightarrow_{\mathcal{P}rg} \Sigma'_1 \cup \Sigma'_2$. Mais évidemment, à partir de $\Sigma_1 \cup \Sigma_2$, on pourrait obtenir des dérivations qui n'auraient aucun sens, notamment si Σ_1 et Σ_2 partagent des variables communes. Or, beaucoup de mes transformations ont pour point de départ l'introduction d'une dérivation "parasite" comme $\Sigma_2 \Rightarrow_{\mathcal{P}rg} \Sigma'_2$ sur une dérivation "initiale" comme $\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma'_1$.

La notion de variable hors-champs permet alors de limiter et de contrôler les interférences de variables. Cette notion est liée au théorème des "ajouts inutiles", qui comme son nom l'indique permet d'ajouter des dérivations parasites qui ne perturbent pas la dérivation initiale.

Définition: Une variable hors-champs est une variable qui ne peut pas apparaître au cours d'une dérivation. Soit donc la fonction φ définie par:

$$\varphi(X) = \{\alpha\} \cup \{y[x] \mid x \in X, \quad y \in \mathcal{V}ars(\mathcal{P}rg)\}$$

une variable z est hors-champs si et seulement si $z \notin$ lfix φ.

La fonction φ traduit la propriété que, si une variable z est introduite au cours d'une dérivation à partir d'un système initial (dans $\mathcal{I}nit(\mathcal{O}bs)$), alors z est soit la variable α, soit une variable introduite par l'application d'une équation du programme $\mathcal{P}rg$ sur une variable x déjà introduite dans la dérivation. Dans ce dernier cas, z est de la forme $y[x]$, avec x une variable déjà introduite et y une variable qui existe dans $\mathcal{P}rg$.

Cas remarquables: une variable de la forme $x.L_m$, où L_m est une nouvelle variable locale (donc non utilisée dans $\mathcal{P}rg$) est hors-champs. Plus généralement, sont également hors-champs les variables de la forme $x.L_m.a$ avec a un attribut quelconque et L_m une nouvelle variable locale. Par ailleurs, une variable de la forme $x.a$ où a est un attribut qui n'est pas mentionné dans le programme est également hors-champs.

On verra dans le théorème dit des "ajouts inutiles" que, f étant une fonction telle que $f(\Sigma) = \{x = t\}$ si x est hors-champs et si $x \notin \mathcal{V}ars(\Sigma)$, alors f est un opérateur d'évaluation partielle.

5.3 Spécialisation

La spécialisation est une transformation d'inspiration fonctionnelle. A la source, on trouve la situation suivante. Considérons une fonction φ, par exemple à deux arguments, et une valeur K (comme Konstante). Je peux avoir envie de définir la fonction ψ suivante:

$$(\forall x) \quad \psi(x) = \varphi(x, K)$$

L'intérêt d'introduire une telle fonction est que l'on peut ensuite l'évaluer partiellement en tenant compte de la valeur connue (K) du deuxième argument de φ. On espère ainsi obtenir des simplifications pour la fonction ψ.

La situation se généralise à plus de deux variables, le cas d'une seule variable étant en partie réglé par évaluation partielle. Du point de vue fonctionnel, cette transformation est facile à définir, du moins dans le cas simple d'une seule et unique spécialisation.

5.3.1 Cas de base

D'après la définition des dépendances approchées, pour tout attribut résultat $r \in \mathcal{R}es$, il existe une fonction φ telle que, si on a:

$$\varphi(v, v_1, \ldots, v_n) = w$$

alors avec $\mathcal{D}ep(r) = \{p_1 \ldots p_n\}$, on a:

$$\left\{ \begin{array}{l} \alpha = v \\ \alpha.p_i = v_i \quad i \leq n \end{array} \right. \Rightarrow_{\mathcal{P}rg} \{\alpha.r = w$$

Dans ce cadre, une spécialisation de la fonction φ par rapport à une de ses variables revient à considérer le cas où l'une des valeurs v_i est une valeur particulière K. Donc, de même que l'on peut définir par exemple une fonction ψ égale à φ quand la valeur d'indice j est égale à K, on peut avoir envie de donner un nom particulier à l'attribut r quand, pour le même exemple, l'attribut p_j reçoit la valeur K.

Soit donc s un nouveau nom d'attribut, supposé en dehors du champs du programme $\mathcal{P}rg$. L'idée est alors de définir s partout où est défini r, la seule différence entre les deux portant sur la valeur du paramètre p_j.

Soit alors le programme suivant, dans lequel $\alpha.L_m$ est une nouvelle variable locale encore jamais utilisée:

$$\mathcal{P}rg' = \bigcup_{c \in \tau_r} \left\{ \begin{array}{lll} c & \to & \alpha.s = \alpha.L_m.r \\ c & \to & \alpha.L_m = \alpha \\ c & \to & \alpha.L_m.p_i = \alpha.p_i \quad i \neq j \\ c & \to & \alpha.L_m.p_j = K \end{array} \right\}$$

Premier principe: les variables $\alpha.s$, $\alpha.L_m$ etc. étant toutes hors-champs, on pourra facilement montrer avec le théorème des ajouts inutiles[1] que la relation $\Rightarrow_{\mathcal{P}rg'}$ est un opérateur d'évaluation partielle.

On peut aussi voir ce résultat sous la forme suivante, qui n'est qu'un extrait de la transformation $\mathcal{T}rans_{EP}$ définie section 5.1:

$$\mathcal{P}rg \cup \mathcal{P}rg' \sim_{\mathcal{O}bs} \mathcal{P}rg$$

Évidemment, les variables hors-champs ne sont pas très intéressantes car elles sont totalement extérieures aux calculs observés dans le champs $\mathcal{O}bs$. Je vais maintenant chercher des attributs hors-champs qui reçoivent, sous certaines conditions, les mêmes valeurs que des attributs "dans" le champs. Deux théorèmes seront utilisés pour ce faire, comme on va le voir tout de suite.

Observons quelques équations obtenues en appliquant l'évaluation partielle. Soit donc un système initial $\Sigma \in \mathcal{I}nit(\mathcal{O}bs)$. Le lecteur vérifiera avec moi que pour toute variable x, si on a:

$$\Sigma(EP)^* \left\{ \begin{array}{ccc} x & = & t \\ x.p_1 & = & t_1 \\ \dots & & \\ x.p_n & = & t_n \end{array} \right\}$$

alors on a aussi, en appliquant l'opérateur $\Rightarrow'_{\mathcal{P}rg}$, la dérivation suivante:

1. Cf. énoncé section 6.1, preuve section 6.5.

$$\Sigma(EP)^* \left\{ \begin{array}{lcll} x.L_m & = & t & \\ x.L_m.p_i & = & t_i & (\forall i \neq j) \\ x.L_m.p_j & = & K & \end{array} \right\}$$

Imaginons maintenant que l'on soit dans le cas particulier où $t_j = K$. On se trouve dans une situation très particulière où l'on a:

$$\left\{ \begin{array}{lcll} x & =_\Sigma & x.L_m & \\ x.p & =_\Sigma & x.L_m.p & (\forall p \in \mathcal{D}ep(r)) \end{array} \right\}$$

Or par définition intuitive de la relation $\mathcal{D}ep$, la connaissance des valeurs des attributs de $\mathcal{D}ep(r)$ suffit à déterminer la valeur de l'attribut r. Ici, $\mathcal{D}ep(r) = \{p\}$, et le théorème de cohérence[2] permet alors d'en déduire la propriété suivante:

$$x.r =_\Sigma x.L_m.r$$

Or, en utilisant les équations de $\mathcal{P}rg'$ on trouve également l'équation suivante:

$$\Sigma \Rightarrow_{\mathcal{P}rg \cup \mathcal{P}rg'} \{x.s = x.L_m.r\}$$

Donc, dans le cas particulier où $x.p_j =_\Sigma K$, on peut déduire aisément $x.r =_\Sigma x.s$.

Deuxième principe: à l'aide du théorème de cohérence, trouver des conditions particulières dans lesquelles des attributs sont égaux, ou plus exactement, des attributs a et b tels que $x.a =_\Sigma x.b$ pour tout x et tout $\Sigma \in \mathcal{I}nit(\mathcal{O}bs)$. Par exemple, ici on a montré que sous la condition $x.p_j =_\Sigma K$ on a $x.r =_\Sigma x.s$.

Troisième principe: La suite de la démarche consiste alors simplement à ajouter ces égalités découvertes avec le second principe. Pour ce faire, il suffit d'introduire le nouvel opérateur $\Rightarrow_{spe}$ défini par:

$$\Sigma \Rightarrow_{spe} \Sigma' \quad \Leftrightarrow \quad \left\{ \begin{array}{ll} (\exists x) & x.p_j = K \in \Sigma \\ \text{et} & \Sigma' = \Sigma \cup \{x.r = x.s\} \end{array} \right\}$$

Le théorème d'extension des égalités[3] montre que cette relation est bien un opérateur d'évaluation partielle, au sens défini à la section 5.1. Il n'y a plus qu'à l'incorporer à la transformation d'évaluation partielle.

5.3.2 Exemple

Prenons un exemple dans lequel on incrémente les éléments d'une liste d'une même quantité:

2. Cf. énoncé section 6.1, preuve section 6.6.
3. Cf. énoncé section 6.1, preuve section 6.4.

$$\mathcal{P}rg = \left\{ \begin{array}{lcl} cons & \rightarrow & \alpha.inc = (cons\ (+\ \alpha.1\ \alpha.p)\ \alpha.2.inc) \\ cons & \rightarrow & \alpha.2.p = \alpha.p \\ nil & \rightarrow & \alpha.inc = (nil) \end{array} \right\}$$

Ainsi, si x est une liste d'entiers, $x.inc$ est la même liste avec tout ses entiers augmentés de $x.p$. On peut vouloir spécialiser ce programme pour le cas où, par exemple, p vaut 1. Mettons que l'on choisisse de nommer *inc1* l'attribut résultant, la transformation de programme s'effectue en deux étapes. Premièrement, je commence par produire un programme additionnel $\mathcal{P}rg'$:

$$\mathcal{P}rg' = \bigcup_{(c \in \{cons, nil\})} \left\{ \begin{array}{lcl} c & \rightarrow & \alpha.inc1 = \alpha.L_1.inc \\ c & \rightarrow & \alpha.L_1 = \alpha \\ c & \rightarrow & \alpha.L_1.p = 1 \end{array} \right\}$$

L'opérateur associé à cette spécialisation est défini par:

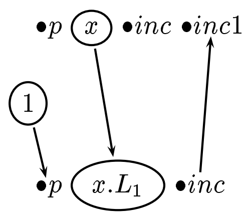
$$\Sigma_1 \ \Rightarrow_{spe} \ \Sigma_2 \Leftrightarrow (\exists x) x.p = 1 \in \Sigma_1 \text{ et } \Sigma_2 = \Sigma_1 \cup \{x.inc = x.inc1\}$$

Je voudrais illustrer deux aspects de la transformation dans cet exemple, son résultat après évaluation partielle, ainsi que le fonctionnement de sa preuve de correction. Je rappelle que pour cette preuve de correction, il suffit de montrer que $\Rightarrow_{spe}$ et $\Rightarrow_{\mathcal{P}rg'}$ sont des opérateurs d'évaluation partielle.

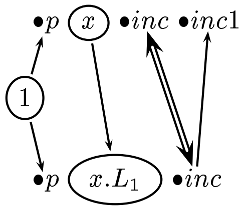
Preuve: j'utilise des diagrammes pour bien montrer les égalités et les équations obtenues. Pour cette illustration, je vais supposer que à partir d'un système initial $\Sigma \in \mathcal{I}nit(\mathcal{O}bs)$ on est parvenu jusqu'à un système dans lequel on a les variables suivantes:

$$\bullet p \ x \ \bullet inc$$

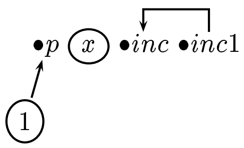
Le premier principe permet d'appliquer les équations de $\mathcal{P}rg'$, qui sont hors-champs, sans changer les résultats des calculs. Donc le premier principe permet d'arriver à:



Alors, imaginons que l'on soit dans le cas où $x.p = 1$, le second principe permet de déduire l'égalité suivante:



Dans ce cas, on a par transitivité l'égalité entre $x.inc$ et $x.inc1$, donc on peut utiliser le troisième principe et incorporer l'équation $x.inc = x.inc1$ dans la dérivation:



Résultat: voyons ce qui se passe après application de l'évaluation partielle.

Commençons par le constructeur *nil*. On obtient par dérivation, entre autres, les équations suivantes (je note entre crochet l'origine de l'équation):

$$\left\{\begin{array}{ccll} \alpha & = & (nil) & \\ \alpha.inc1 & = & \alpha.L_1.inc & [\mathcal{P}rg'] \\ \alpha.L_1 & = & (nil) & [\mathcal{P}rg'] \\ \alpha.L_1.inc & = & (nil) & [\mathcal{P}rg] \\ \alpha.inc1 & = & (nil) & [\text{subst.}] \end{array}\right\}$$

Rien du bien surprenant, dans le cas *nil*, les deux attributs *inc* et *inc1* sont identiques. Du côté du constructeur *cons* c'est un tout petit peu plus compliqué, mais on a entre autres les équations suivantes:

$$\left\{\begin{array}{ccll} \alpha & = & (cons\ \alpha.1\ \alpha.2) & \\ \alpha.inc1 & = & \alpha.L_1.inc & [\mathcal{P}rg'] \\ \alpha.L_1.p & = & 1 & [\mathcal{P}rg'] \\ \alpha.L_1 & = & (cons\ \alpha.1\ \alpha.2) & [\mathcal{P}rg'] \end{array}\right\}$$

Maintenant, si on poursuit l'évaluation partielle en appliquant le programme $\mathcal{P}rg$ sur $\alpha.L_1$, on a:

$$\left\{\begin{array}{ccll} \alpha.L_1.inc & = & (cons\ (+\ \alpha.L_1.1\ \alpha.L_1.p)\ \alpha.L_1.2.inc) & [\mathcal{P}rg] \\ \alpha.L_1.2.p & = & \alpha.L_1.p & [\mathcal{P}rg] \\ \alpha.L_1.1 & = & \alpha.1 & [\text{subst.}] \\ \alpha.L_1.2 & = & \alpha.2 & [\text{subst.}] \end{array}\right\}$$

Entre autre, on en tire les équations suivantes qui peuvent nous intéresser pour le calcul de $\alpha.inc1$:

$$\left\{\begin{array}{lcll} \alpha.\mathit{inc1} & = & (\mathit{cons}\ (+\ \alpha.1\ 1)\ \alpha.L_1.2.\mathit{inc}) & [\text{subst.}] \\ \alpha.L_1.2.p & = & 1 & [\text{subst.}] \end{array}\right\}$$

Ici, on peut appliquer $\Rightarrow_{spe}$ car on a $x.p = 1$ avec dans le cas présent $x = \alpha.L_1.2$. Cela donne au final le jeu d'équations suivant:

$$\left\{\begin{array}{ccll} \alpha.L_1.2.\mathit{inc} & = & \alpha.L_1.2.\mathit{inc1} & [\Rightarrow_{spe}] \\ \alpha.\mathit{inc1} & = & (\mathit{cons}\ (+\ \alpha.1\ 1)\ \alpha.L_1.2.\mathit{inc1}) & [\text{subst.}] \end{array}\right\}$$

Si on utilise l'opérateur d'évaluation partielle implémenté au chapitre 7, on obtient le programme suivant:

$$\mathcal{P}rg\text{''} = \mathcal{P}rg \cup \left\{\begin{array}{ccl} \mathit{cons} & \rightarrow & \alpha.\mathit{inc1} = (\mathit{cons}\ (+\ \alpha.1\ 1)\ \alpha.2.\mathit{inc1}) \\ \mathit{nil} & \rightarrow & \alpha.\mathit{inc1} = (\mathit{nil}) \end{array}\right\}$$

La seule différence vient de l'abandon du $\alpha.L_1.2$ au profit de $\alpha.2$. En effet, c'est une transformation possible car $\alpha.L_1 = \alpha.2$ et qu'il n'y a pas encore d'équation sur $\alpha.2.\mathit{inc1}$ dans le programme $\mathcal{P}rg \cup \mathcal{P}rg'$.

5.3.3 Cas général

Dans le cas général, il peut y avoir plus d'une spécialisation d'un même argument, mais sur des arguments différents. Le tout peu vite devenir compliqué si l'on veut éviter les redondances. Du point de vue théorique, cela ne change rien, mais du côté de l'implémentation, la complexité s'en ressent. Le lecteur intéressé par ce problème peut se référer à la section 5.9.1 pour trouver plus de détails à ce sujet.

5.4 Composition

Encore une fois, c'est une idée fonctionnelle qui est à l'origine de cette transformation. Soient donc deux fonctions f et g, l'une d'arité $n+1$, l'autre d'arité m. Je considère ici une composition de f et g, plus précisément, la fonction h définie par:

$$h(x_1, \ldots, x_n, y_1, \ldots, y_m) = f(x_1, \ldots, x_n, g(y_1, \ldots, y_m))$$

On espère alors que la fonction f puisse tirer partie de la définition de la fonction g dans certains cas. Par exemple, dans le cas des fonctions `length` et `inc`, on espère trouver une fonction h qui ait en définitive la même définition que `length`.

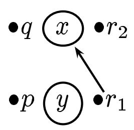
Remarque: cette notion de composition existe aussi pour les grammaires attribuées [8, 6, 30] et elle est même plus puissante [3]. Cependant, je n'ai pas pu introduire directement ces techniques de composition à cause notamment des variables locales qui introduisent des problèmes en cas de non-linéarité. Ma démarche, originale, a été de couper en deux la composition des grammaires attribuées pour en isoler une partie commune avec la composition fonctionnelle. La seconde partie sera vue à la section suivante. Je ferai au cours de cette section une autre remarque sur le sujet au moment opportun.

5.4.1 Cas de base

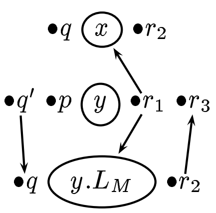
D'un point de vue équationnel, il faut considérer deux attributs résultats r_1 et r_2. Les attributs paramètres dont ils peuvent dépendre sont $\mathcal{D}ep(r_1) = \{p_1 \dots p_n\}$ et $\mathcal{D}ep(r_2) = \{q_1 \dots q_m\}$. La composition de $r1$ et de r_2 peut se définir comme le calcul d'un attribut r_3, résultat de r_2 sur la valeur calculée sur r_1. Bien entendu, r_3 est susceptible de dépendre des attributs de $\mathcal{D}ep(r_1)$, mais également de ceux de $\mathcal{D}ep(r_2)$. Pour éviter des collision de noms, il vaut toutefois mieux renommer au passage les attributs de $\mathcal{D}ep(r_2)$. On se donne donc des attributs $q'_1, \dots, q'_m$ qui soient totalement nouveaux (donc hors-champs). Il est alors correct d'ajouter au programme $\mathcal{P}rg$ le programme additionnel suivant:

$$\mathcal{P}rg' = \bigcup_{c \in \tau_{\{r_1\}}} \left\{ \begin{array}{lll} c & \rightarrow & \alpha.r_3 = \alpha.L_M.r_2 \\ c & \rightarrow & \alpha.L_M.q_j = \alpha.q'_j \quad (j \leq m) \\ c & \rightarrow & \alpha.L_M = \alpha.r_1 \end{array} \right\}$$

Bien entendu, L_M est un nouveau nom de variable locale pour que les équations soient hors-champs. Alors, on peut vérifier que, si au cours d'une dérivation on a $x = y.r_1$, alors pour calculer $x.r_2$, on peut aussi bien calculer $y.r_3$ avec $y.q'_j = x.q_j$ pour tout $j \leq m$. Pour illustrer ceci, je vais utiliser mes petits diagrammes, en choisissant le cas $n = m = 1$. On a donc au départ:



En utilisant le premier principe, il est correct d'ajouter les équations hors-champs suivantes, issues du programme $\mathcal{P}rg'$:



On voit alors sur ce diagramme que, si on pouvait ajouter une équation $y.q' = x.q$, on aurait grâce au deuxième principe l'équation $x.r_2 =_\Sigma y.r_3$. Le problème est que l'on ne peut pas ajouter librement l'équation $y.q' = x.q$, car maintenant que $\mathcal{P}rg'$ a été ajouté à $\mathcal{P}rg$, cette équation n'est plus hors-champs. En fait, il se pourrait que l'on ait une autre variable x' telle que $x' = y.r_1$, et alors il pourrait y avoir une collision entre des valeurs différentes sur $y.q'$.

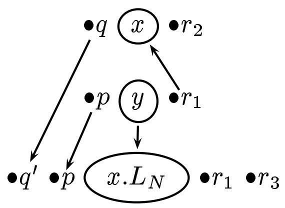
Qu'à cela ne tienne, il suffit auparavant d'effectuer une copie "propre" du calcul de r_1 sur y sur la nouvelle variable locale $x.L_N$. Je définis alors l'extension de relation $\rightarrow_{compo}$ définie par:

$$\Sigma_1 \;\rightarrow_{compo}\; \Sigma_2 \Leftrightarrow$$
$$(\exists x, y) \begin{cases} x = y.r_1 \in \Sigma_1 \\ \Sigma_2 = \Sigma_1 \cup \left\{ \begin{array}{ccll} x.r_2 & = & x.L_N.r_3 & \\ x.L_N & = & y & \\ x.L_N.p_i & = & y.p_i & (i \leq n) \\ x.L_N.q'_j & = & x.q_j & (j \leq m) \end{array} \right\} \end{cases}$$

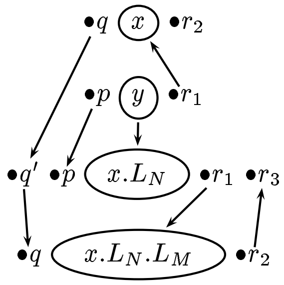
La variable $x.L_N$ sera fraîche à chaque nouvelle utilisation de $\rightarrow_{compo}$, donc systématiquement hors-champs.

Remarque: par rapport à la composition classique des grammaires attribuées, ce genre de considération est assez original, et vient du fait que les programmes équationnels ont un pouvoir d'expression plus grand. En particulier, une même variable peut être utilisée plusieurs fois, par exemple lors d'un calcul non-linéaire. De ce fait, faire une copie "neuve" de la variable y correspond tout simplement à faire un nouvel appel de fonction, donc dans un contexte isolé du reste (d'où la notion de variable "hors-champs"). C'est pourquoi cette transformation est plutôt d'inspiration fonctionnelle. On verra dans la transformation suivante que l'on ne pourra pas recourir à ce genre d'astuce et que la transformation sortira alors du cadre fonctionnel classique.

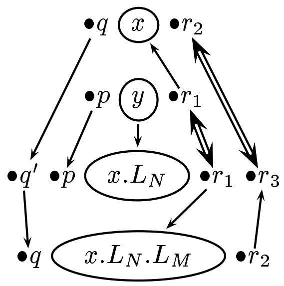
Je reprends mes petits diagrammes. Partant du même point, il est maintenant correct d'ajouter les équations suivantes en vertu du premier principe:



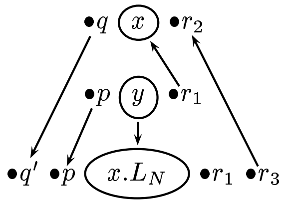
Maintenant, en appliquant le programme $\mathcal{P}rg'$, on trouve des équations que je peux représenter par:



Je vais maintenant utiliser deux fois le second principe, pour ajouter des égalités. Tout d'abord, on constate que $x.L_n.r_1 =_\Sigma y.r_1$ et ensuite on en déduit par transitivité que $x.r_2 =_\Sigma x.L_N.r_3$.



Ce qui permet de justifier toutes les équations introduites par $\rightarrow_{compo}$, que je schématise avec le diagramme suivant:



5.4.2 Exemple

Je reprends l'exemple vu pour la spécialisation, et compose ce programme avec celui de la longueur d'une liste:

$$
\begin{array}{rcl}
\mathcal{P}rg & = & \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.inc = (cons\ (+\ \alpha.1\ \alpha.p)\ \alpha.2.inc) \\ cons & \rightarrow & \alpha.2.p = \alpha.p \\ nil & \rightarrow & \alpha.inc = (nil) \end{array} \right\} \\
& \cup & \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.length = (+\ 1\ \alpha.2.length) \\ nil & \rightarrow & \alpha.length = 0 \end{array} \right\}
\end{array}
$$

Maintenant, imaginons que je veuille composer *inc* et *length*, donc calculer la longueur d'une liste incrémentée. Je choisi de nommer r l'attribut résultant. En appliquant cette transformation, j'obtiens tout d'abord le programme additionnel suivant:

$$\mathcal{P}rg' = \bigcup_{c \in \{cons, nil\}} \left\{ \begin{array}{lll} c & \rightarrow & \alpha.r = \alpha.L_M.length \\ c & \rightarrow & \alpha.L_M = \alpha.inc \end{array} \right\}$$

Après évaluation partielle, on va obtenir des simplifications intéressantes. Voyons donc en premier lieu se qui se passe pour le constructeur *nil*. Voici quelques équations produites par l'évaluation partielle. Dans un premier temps, l'opérateur $\Rightarrow_{\mathcal{P}rg'}$ est utilisé, ce qui revient à appliquer les équations du programme $\mathcal{P}rg'$:

$$\left\{ \begin{array}{ccll} \alpha & = & (nil) & \\ \alpha.r & = & \alpha.L_M.length & [\mathcal{P}rg'] \\ \alpha.L_M & = & \alpha.inc & [\mathcal{P}rg'] \end{array} \right\}$$

Puis, on peut appliquer le programme définissant l'attribut *inc* sur α, puis l'attribut *length* par dessus:

$$\left\{ \begin{array}{ccll} \alpha.inc & = & (nil) & [\mathcal{P}rg] \\ \alpha.L_M & = & (nil) & [\text{subst.}] \\ \alpha.L_M.length & = & 0 & [\mathcal{P}rg] \\ \alpha.r & = & 0 & [\text{subst.}] \end{array} \right\}$$

Pour le constructeur *cons*, les choses sont un peu plus compliquées, mais avec le même schéma on obtient dans un premier temps:

$$\left\{ \begin{array}{ccll} \alpha & = & (cons\ \alpha.1\ \alpha.2) & \\ \alpha.r & = & \alpha.L_M.length & [\mathcal{P}rg'] \\ \alpha.L_M & = & \alpha.inc & [\mathcal{P}rg'] \end{array} \right\}$$

Et dans un second temps:

$$\left\{ \begin{array}{ccll} \alpha.inc & = & (cons\ (+\ \alpha.p\ \alpha.1)\ \alpha.2.inc) & [\mathcal{P}rg] \\ \alpha.L_M & = & (cons\ (+\ \alpha.p\ \alpha.1)\ \alpha.2.inc) & [\text{subst.}] \\ \alpha.2.p & = & \alpha.p & [\mathcal{P}rg] \\ \alpha.L_M.length & = & (+\ 1\ \alpha.L_M.2.length) & [\mathcal{P}rg] \\ \alpha.L_M.2 & = & \alpha.2.inc & [\text{subst.}] \end{array} \right\}$$

A ce stade, on a alors la possibilité de continuer l'évaluation partielle avec l'opérateur $\rightarrow_{compo}$, dans la mesure où l'on a une équation $x = y.inc$ et que l'on veut $x.length$, avec dans le cas présent $x = \alpha.L_M.2$ et $y = \alpha.2$. On trouve alors les équations suivantes:

$$\left\{ \begin{array}{ccll} \alpha.L_M.2.length & = & \alpha.L_M.2.r & [\rightarrow_{compo}] \\ \alpha.L_M.2 & = & \alpha.2 & [subst.] \\ \alpha.r & = & (+\ 1\ \alpha.L_M.2.r) & [subst.] \end{array} \right\}$$

En utilisant l'OEP défini au chapitre 7, on obtient pour finir le programme suivant:

$$\mathcal{P}rg" = \mathcal{P}rg \cup \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.r = (+\ 1\ \alpha.2.r) \\ nil & \rightarrow & \alpha.r = 0 \end{array} \right\}$$

Ce qui est exactement ce que l'on veut! Remarquez au passage que à priori, l'attribut r aurait pu dépendre de p, puisque $\mathcal{D}ep(inc) = \{p\}$. D'ailleurs, cet attribut est bien présent au cours de l'évaluation partielle. Mais à la fin, on se rend compte que cette dépendance peut disparaître.

5.4.3 Cas général

Dans le cas général, on peut tout à fait avoir plus d'une composition. Or le principal problème est de se demander si on aura l'associativité. Informellement, on s'attend à ce que l'on ait:

$$f_1 \circ (f_2 \circ f_3) = (f_1 \circ f_2) \circ f_3$$

Malheureusement, avec ce qui est décrit dans les sections précédentes, on aura bien l'égalité des deux calculs, mais on aura pas l'identité syntaxique: à chacune des deux compositions correspondra un attribut différent. Du côté théorique, cela ne change presque rien, mais on risque de perdre des opportunités d'utiliser le second principe (trouver des attributs identiques). Côté implémentation, cela risque de demander des calculs inutiles et encombrant. Le lecteur intéressé par ce problème peut se reporter à la section 5.9.2 où il trouvera une méthode judicieuse pour contourner ces difficultés.

5.5 Précalculation

A propos de la composition, il faut tout de même signaler qu'elle ne donne pas toujours des résultats satisfaisants. Par exemple, essayons de composer l'inversion de liste avec le calcul de la longueur, histoire de rester en terrain connu. Je note r l'attribut résultant de la composition et l'on aura *a priori* les dépendances suivantes:

$$\mathcal{D}ep(r) \subset \mathcal{D}ep(length) \cup \mathcal{D}ep(rev)$$

En particulier, on aura donc $\mathcal{D}ep(r) \subset \{rev\_h\}$. Voyons maintenant le résultat de cette composition, de moins celui obtenu après évaluation partielle:

$$\mathcal{P}rg = \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.r = \alpha.2.r \\ " & \rightarrow & \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ nil & \rightarrow & \alpha.r = \alpha.L_M.length \\ " & \rightarrow & \alpha.L_M = \alpha.rev\_h \end{array} \right\}$$

On voit ici que l'on est loin de la simplification attendue. Comme je le disais dans l'introduction de la thèse, il y a bien eu une petite transformation, mais elle est minime: le calcul de la composition de *length* et *rev* à été remplacé par le calcul de l'attribut r (ce qui est correct car la composition est correcte); mais, on y a pas gagné grand chose, car le calcul de la longueur de la liste inversée est toujours présent et il a juste été déplacé au niveau du constructeur *nil*.

Remarque: en programmation fonctionnelle, ce programme correspond à la fonction `revlength` définie par:

```
let revlength h = fun
  cons a b -> revlength (cons a h) b
| nil -> length h
```

On voit mieux sous cette forme que le calcul de la longueur de la liste a simplement été déplacé au niveau du constructeur *nil*. Je reviens maintenant au programme équationnel $\mathcal{P}rg$

On détecte alors une composition un peu particulière, à savoir le calcul d'un attribut résultat (*length*) sur un attribut paramètre (*rev_h*). Comme on n'est pas contraint par le nombre et la qualité des attributs paramètres, je vais pouvoir utiliser une astuce originale pour effectuer ce type de composition.

Cette transformation consiste tout simplement à précalculer la valeur de la composition au niveau où le paramètre est calculé. C'est une forme de composition un peu plus complexe que la composition de deux attributs résultats et qu'il sera assez difficile de coder directement au niveau fonctionnel (pour le cas général).

5.5.1 Premier exemple

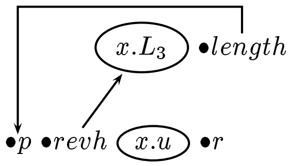
Je reviens à ma composition de l'inversion et de la longueur. Pour clarifier la suite, je suppose que ce programme contient aussi un constructeur *main*, dont l'attribut résultat *call* calcule le résultat de la composition sur la liste passée par l'attribut paramètre *arg*:

$$\left\{\begin{array}{lll} main & \rightarrow & \alpha.call = \alpha.L_1.r \\ " & \rightarrow & \alpha.L_1 = \alpha.arg \\ " & \rightarrow & \alpha.L_1.rev\_h = (nil) \\ \ldots & & \end{array}\right\}$$

Le programme total, que je note $\mathcal{P}rg$ à partir de maintenant est donc:

$$\mathcal{P}rg = \left\{\begin{array}{lll} main & \rightarrow & \alpha.call = \alpha.L_1.r \\ " & \rightarrow & \alpha.L_1 = \alpha.arg \\ " & \rightarrow & \alpha.L_1.rev\_h = (nil) \\ cons & \rightarrow & \alpha.r = \alpha.2.r \\ " & \rightarrow & \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ nil & \rightarrow & \alpha.r = \alpha.L_M.length \\ " & \rightarrow & \alpha.L_M = \alpha.rev\_h \end{array}\right\}$$

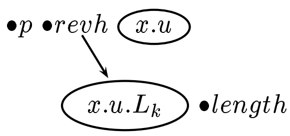
Imaginons alors que, à chaque fois que l'attribut paramètre *rev_h* est défini, un nouvel attribut p soit également défini. Cet attribut sera par définition toujours égal à la longueur de la liste contenue par l'attribut *rev_h*. Pour illustrer cette idée, je vais utiliser un petit diagramme. Je suppose donc que l'on est en train de calculer la valeur de $y.r$ en fonction de $y.p$ sur une variable y. Puisque l'on doit rajouter un calcul sur le calcul courant, je vais donc supposer que y est de la forme $x.u$, ou u est soit "$.k$", soit "$.L_k$" avec k quelconque. Pour précalculer une valeur, il suffit d'ajouter une nouvelle variable locale sur x, par exemple $x.L_N$:



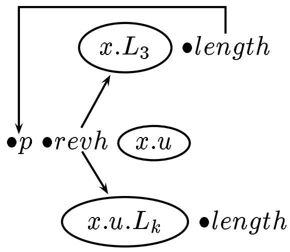
Ces ajouts sont corrects car les équations concernées sont toutes hors-champs (premier principe). On peut donc définir un nouvel opérateur d'évaluation partiel de la forme $\Rightarrow_{\mathcal{P}rg'}$ où $\mathcal{P}rg'$ est le programme suivant:

$$\mathcal{P}rg' = \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.2.p = \alpha.L_N.length \\ " & \rightarrow & \alpha.L_N = \alpha.2.rev\_h \\ main & \rightarrow & \alpha.L_1.p = \alpha.L_N.length \\ " & \rightarrow & \alpha.L_N = \alpha.L_1.rev\_h \end{array} \right\}$$

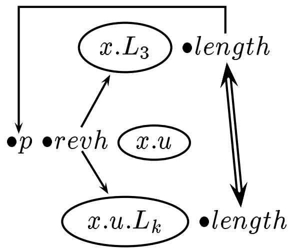
Voyons alors ce qui se passe quand on a besoin de calculer la longueur du paramètre $x.u.rev\_h$. Cela signifie que auparavant, la valeur de ce paramètre aura été stockée dans une variable local, par exemple $x.u.L_k$. On cherche donc la valeur de $x.u.L_k.length$, situation que l'on peut illustrer par le diagramme suivant:



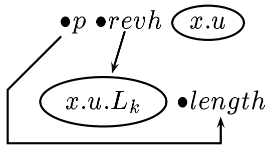
Si maintenant on ajoute (premier principe) les équations du programme $\mathcal{P}rg'$, on obtient le diagramme suivant:



On a une opportunité d'utiliser le second principe, et de découvrir une égalité:



Par transitivité, on peut donc déduire l'égalité de $x.u.p$ et $x.u.L_k.length$, et donc avec le troisième principe, il est correcte d'ajouter l'équation suivante:



Revenons au programme $\mathcal{P}rg \cup \mathcal{P}rg'$ et appliquons lui ces idées. Après une étape d'évaluation partielle, on trouve alors le programme suivant:

$$\left\{\begin{array}{lll} main & \rightarrow & \alpha.call = \alpha.L_1.r \\ " & \rightarrow & \alpha.L_1.p = 0 \\ " & \rightarrow & \alpha.L_1 = \alpha.arg \\ cons & \rightarrow & \alpha.r = \alpha.2.r \\ " & \rightarrow & \alpha.2.p = (+\ 1\ \alpha.L_N.2.length) \\ " & \rightarrow & \alpha.L_N.2 = \alpha.rev\_h \\ \ldots & & \end{array}\right\}$$

Et maintenant, on utilise le troisième principe pour relier la longueur de $\alpha.rev\_h$ avec $\alpha.p$. Ce qui donnerai le programme suivant:

$$\mathcal{P}rg \sim_{\mathcal{O}bs} \left\{\begin{array}{lll} main & \rightarrow & \alpha.call = \alpha.L_1.r \\ " & \rightarrow & \alpha.L_1.p = 0 \\ " & \rightarrow & \alpha.L_1 = \alpha.arg \\ cons & \rightarrow & \alpha.r = \alpha.2.r \\ " & \rightarrow & \alpha.2.p = (+\ 1\ \alpha.p) \\ nil & \rightarrow & \alpha.r = \alpha.p \end{array}\right\}$$

Ce qui, assurément, correspond à une composition réussie.

Remarque: je signale que l'on peut arriver au même résultat en restant au niveau fonctionnel, en introduisant la fonction `h` telle que:

```
let h p hr x = revlength hr x
```

et en utilisant les deux règles de transformation suivantes: dans l'évaluation partielle de `h`, on peut remplacer `(length h)` par `p`, et dans les autres évaluations partielles, on peut remplacer `(revlength hr x)` par `(h (length hr) hr x)`.

Mais tout marche bien parce que la fonction `length` est d'arité 1. Ce genre d'astuce ne marche plus pour des arités plus grandes, comme on va le voir dans l'exemple qui suit.

5.5.2 Deuxième exemple

Je voudrais maintenant vous montrer ce qui se passe si l'on cherche à composer, par exemple, l'inversion et l'incrémentation. Rappelons le programme équationnel correspondant:

$$\begin{array}{rl}\mathcal{P}rg_0 = & \left\{\begin{array}{ccl} cons & \rightarrow & \alpha.rev = \alpha.2.rev \\ cons & \rightarrow & \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ nil & \rightarrow & \alpha.rev = \alpha.rev\_h \end{array}\right\} \\ \cup & \left\{\begin{array}{ccl} cons & \rightarrow & \alpha.inc = (cons\ (+\ \alpha.1\ \alpha.p)\ \alpha.2.inc) \\ cons & \rightarrow & \alpha.2.p = \alpha.p \\ nil & \rightarrow & \alpha.inc = (nil) \end{array}\right\}\end{array}$$

Pour compléter ce programme, je rajoute un constructeur *main* pour coder la fonction qui inverse et incrémente une liste. L'attribut résultat *call* sera le résultat de la fonction. Il faut lui adjoindre deux attributs paramètres: *arglist* pour la liste à traiter et *arginc* pour la valeur de l'incrémentation.

$$\mathcal{P}rg = \left\{\begin{array}{ccl} main & \rightarrow & \alpha.call = \alpha.L_1.inc \\ " & \rightarrow & \alpha.L_1 = \alpha.L_2.rev \\ " & \rightarrow & \alpha.L_1.p = \alpha.arginc \\ " & \rightarrow & \alpha.L_2 = \alpha.arglist \\ " & \rightarrow & \alpha.L_2.rev\_h = (nil) \\ \ldots & & \end{array}\right\}$$

Commençons par une composition – sans problème – des attributs *rev* et *inc*. J'appelle r_1 l'attribut résultat correspondant. On a, *a priori*, $\mathcal{D}ep(r_1) \subset \mathcal{D}ep(rev) \cup \mathcal{D}ep(inc)$. Mais pour éviter les collisions de noms, je rappelle (voir section 5.4) que l'on renomme les attributs de $\mathcal{D}ep(inc)$. L'attribut p étant ici renommé en p_1, on aura:

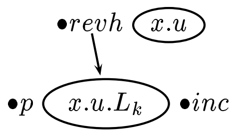
$$\mathcal{D}ep(r_1) \subset \{p_1\ rev\_h\}$$

Après la transformation de composition (et l'évaluation partielle bien entendu) on obtient le programme suivant, que je note $\mathcal{P}rg$ (je ne répète pas les équations de programme précédentes pour des raisons de lisibilité):

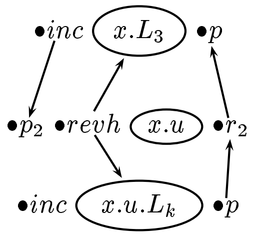
$$\mathcal{P}rg = \mathcal{P}rg_0 \cup \left\{ \begin{array}{ccl} cons & \rightarrow & \alpha.r_1 = \alpha.2.r_1 \\ \text{"} & \rightarrow & \alpha.2.p_1 = \alpha.p_1 \\ \text{"} & \rightarrow & \alpha.2.rev\_h = \alpha.rev\_h \\ nil & \rightarrow & \alpha.r_1 = \alpha.L_N.inc \\ \text{"} & \rightarrow & \alpha.L_N = \alpha.rev\_h \\ \text{"} & \rightarrow & \alpha.L_N.p = \alpha.p_1 \\ main & \rightarrow & \alpha.call = \alpha.L_2.r_1 \\ \text{"} & \rightarrow & \alpha.L_2 = \alpha.arglist \\ \text{"} & \rightarrow & \alpha.L_2.p_1 = \alpha.arglist \\ \text{"} & \rightarrow & \alpha.L_2.rev\_h = (nil) \end{array} \right\}$$

Comme dans le premier exemple, on trouve pour le constructeur *nil* une composition de l'attribut *inc* sur l'attribut *rev_h*. Encore une fois, l'idée de le transformation va être de précalculer cette composition au même niveau que celui du calcul de l'attribut *rev_h*. Mais ici, il faudra au préalable remonter à ce même niveau la valeur du paramètre p nécessaire au calcul de *inc*.

Pour illustrer l'idée de cette précalculation, prenons une situation on l'on calcule l'attribut *inc* sur la variable $x.u.rev\_h$ comme dans le premier exemple. On a donc la situation suivante:

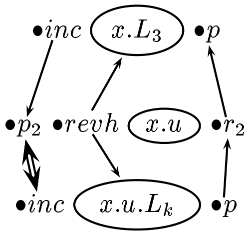


Pour remonter le calcul "au-dessus" de $x.u$, je vais donc utiliser deux nouveaux attributs hors-champs et une nouvelle variable locale $x.L_N$. Le premier attribut, noté p_2, servira pour transporter l'attribut *inc* qui aura été calculé sur la valeur de *rev_h*. Le deuxième attribut, noté r_2, transportera le paramètre p correspondant. Cette idée est illustrée par le diagramme suivant:

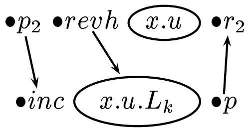


Remarque: dans ce diagramme, les conventions pour placer les attributs *inc* et p ont été inversées pour faciliter la lecture du diagramme. Entre autres remarques, il n'y a pas de dépendance fonctionnelle entre les attributs r_2 et p_2. C'est un fait nouveau qui montre bien que ces calculs ne seront pas facilement modélisés par une fonction. Pour retrouver des fonctions, il faudra calculer des stratégies et on verrait alors que r_2 et p_2 interviennent dans des fonctions bizarres qu'il eut été difficile d'imaginer au premier abord.

En utilisant le second principe, on démontre alors l'égalité du diagramme suivant:



Ce qui permet alors d'utiliser le troisième principe pour obtenir:



Pour le programme $\mathcal{P}rg$, ces idées s'appliquent en définissant un opérateur d'évaluation partielle de la forme $\Rightarrow_{\mathcal{P}rg'}$ où $\mathcal{P}rg'$ est le programme additionnel suivant:

$$\mathcal{P}rg' = \bigcup_{(c,x)\in A} \left\{ \begin{array}{lcl} c & \rightarrow & x.p_2 = \alpha.L_N.inc \\ " & \rightarrow & \alpha.L_N = x.rev\_h \\ " & \rightarrow & \alpha.L_N.p = x.r_2 \end{array} \right\}$$

en notant A l'ensemble des couples (c, x) sur lesquels il faut précalculer inc et p, c'est à dire tels qu'il existe un terme t pour lequel $c \rightarrow x.rev\_h = t$ est une équation du programme. Ici, on a donc $A = \{(cons, \alpha.2)\ (main, \alpha.L_2)\}$.

Maintenant, il faut pouvoir relier les attributs comme il faut. Chaque fois que l'on se trouve dans une situation où $y = x.rev\_h$, on peut essayer d'utiliser le calcul de r_2 et p_2 (sur x) à la place du calcul de inc et p (sur y). Toutefois, on ne peut pas toujours incorporer librement ces égalités. En effet, il faut s'assurer que la variable $x.r_2$ est hors-champs et qu'elle n'est pas déjà présente dans le système. Ce n'est pas évident et il se pourrait, par exemple, qu'il existe une autre variable $z = x.rev\_h$ telle que $x.r_2 = z$.

Ce n'est pas un problème difficile. L'ajout de cette nouvelle équation étant effectuée par un opérateur d'évaluation partielle noté $\rightarrow_{prec}$, il suffit de contrôler que ce dernier ne sera utilisée qu'une seule fois sur chaque variable. L'opérateur est définis ainsi :

$$\Sigma_1 \ \rightarrow_{prec} \ \Sigma_2 \Leftrightarrow \left\{ \begin{array}{l} y = x.rev\_h \in \Sigma_1 \\ \Sigma_2 = \Sigma_1 \cup \left\{ \begin{array}{c} y.inc = x.p_2 \\ x.r_2 = y.p \end{array} \right\} \end{array} \right.$$

Au final, on obtient après une étape d'évaluation partielle le programme suivant :

$$\left\{\begin{array}{lcl} cons & \rightarrow & \alpha.r_1 = \alpha.2.r_1 \\ " & \rightarrow & \alpha.2.p_1 = \alpha.p_1 \\ " & \rightarrow & \alpha.2.p_2 = (cons\ (+\ \alpha.1\ \alpha.2.r_2)\ \alpha.p_2) \\ " & \rightarrow & \alpha.r_2 = \alpha.2.r_2 \\ nil & \rightarrow & \alpha.r_1 = \alpha.p_2 \\ " & \rightarrow & \alpha.r_2 = \alpha.p_1 \\ main & \rightarrow & \alpha.call = \alpha.L_2.r_1 \\ " & \rightarrow & \alpha.L_2 = \alpha.arglist \\ " & \rightarrow & \alpha.L_2.p_1 = \alpha.arglist \\ " & \rightarrow & \alpha.L_2.p_2 = (nil) \end{array}\right\}$$

Le lecteur pourra alors vérifier qu'une élimination des identités (voir transformation 5.6) détecte que les attributs r_2 et p_1 sont toujours égaux. Ce qui permet alors d'obtenir le programme suivant:

$$\left\{\begin{array}{lcl} cons & \rightarrow & \alpha.r_1 = \alpha.2.r_1 \\ " & \rightarrow & \alpha.2.p_1 = \alpha.p_1 \\ " & \rightarrow & \alpha.2.p_2 = (cons\ (+\ \alpha.1\ \alpha.p_1)\ \alpha.p_2) \\ nil & \rightarrow & \alpha.r_1 = \alpha.p_2 \\ main & \rightarrow & \alpha.call = \alpha.L_2.r_1 \\ " & \rightarrow & \alpha.L_2 = \alpha.arglist \\ " & \rightarrow & \alpha.L_2.p_1 = \alpha.arglist \\ " & \rightarrow & \alpha.L_2.p_2 = (nil) \end{array}\right\}$$

Encore une fois, c'est une composition bien réussie qui ne fait que les calculs strictement nécessaires.

Remarque: il est pour ainsi dire impossible de traduire cette transformation dans un cadre purement fonctionnel. Le problème vient précisément du fait que l'on ajoute des calculs sur x pour avoir un effet sur $x.u$. Autrement dit, on doit faite remonter des calculs le long des paramètres, au niveau où ces derniers sont définis. En grammaires attribuées, cette transformation est inclue dans la composition, mais elle ne pourrait pas s'appliquer telle qu'elle en raison des problèmes de linéarité.

D'un point de vue fonctionnel, ces transformations permettent d'obtenir des transformations de programmes encore jamais obtenues. Je me permet de renvoyer le lecteur au chapitre 4 pour y trouver d'autres exemples.

5.6 Élimination d'identité

Dans de nombreux cas, les transformations introduisent des attributs qui, par le jeu de certaines évaluations partielles, ne sont que des identités. D'une manière plus générale, si on prouve que pour une fonction f on a par exemple:

$$(\forall x, y) \quad f(x,y) = y$$

alors on a tout intérêt à remplacer le calcul de $f(x,y)$ par y. En sémantique équationnelle, cette propriété correspond à trouver un attribut paramètre p et un attribut résultat r tels que l'on ait:

$$(\forall \Sigma \in \mathcal{I}nit(\mathcal{O}bs))(\forall x) \quad x.r =_\Sigma x.p$$

Définition: je note $p \sim r$ cette propriété.

Si l'on a $p \sim r$, alors il est correct, grâce au théorème d'extension des égalités[4], d'ajouter des équations de la forme $x.r = x.p$ avec x quelconque à n'importe quelle étape de dérivation.

Pour trouver les attributs correspondants, je vais utiliser un raisonnement par récurrence sur les dérivations. L'idée intuitive de ces récurrences est que, si pour tout $x \in \mathcal{X}_{loc}$ on a $x.r = x.p$ et que l'on peut en déduire que $\alpha.r = \alpha.p$, alors on peut affirmer que $p \sim r$. Plus précisément, on peut chercher à prouver que:

$$\{\alpha = (c\ \alpha.1 \ldots \alpha.n)\} \cup \{x.r = x.p \mid x \in \mathcal{X}_{loc}\} \Rightarrow_{\mathcal{P}rg} \{\alpha.r = \alpha.p\}$$

En pratique, je cherche en parallèle l'ensemble I de tous les couples (p,r) tels que $p \sim r$. Je cherche donc le plus grand point fixe de la fonction Ψ définie ci-dessous:

$$\begin{array}{l}\Psi(I) = \{(p,r) \in I \mid (\forall c \in \tau_r) \\ \quad \left(\begin{array}{ll} & \{\alpha = (c\ \alpha.1 \ldots \alpha.n)\} \\ \cup & \{x.r' = x.p' \mid x \in \gamma_c,\ (r',p') \in I\}\end{array}\right) \Rightarrow_{\mathcal{P}rg} \{\alpha.r = \alpha.p\}\ \}\end{array}$$

Alors, avec une preuve qui est une récurrence intuitivement simple mais techniquement compliquée (Cf. section 6.7) on montre que :

$$(p,r) \in \text{gfix}\ \Psi \quad \Rightarrow \quad p \sim r$$

Pour calculer le point fixe I_∞, on peut bien entendu utiliser les résultats de la section 10.2, à savoir:

$$\begin{array}{l}\text{gfix}\ \Psi = \text{fix}\ \Psi\ I_0 \\ I_0 = \{(p,r) \mid p \in \mathcal{P}rm, r \in \mathcal{R}es\}\end{array}$$

En fait on peut faire mieux, car il suffit de prendre pour I_0 un ensemble de couple (p,r) plausibles. Je propose l'ensemble suivant:

$$I_0 = \{(p,r) \mid (\exists c)\ c \to \alpha.r = \alpha.p \in \mathcal{P}rg\}$$

En effet, on a nécessairement avec cet ensemble I_0 la propriété $I_0 \supset I_\infty$.

En utilisant le théorème d'extension des égalités, on prouve alors que la relation $\to_{id}$ définie par:

4. Cf. énoncé section 6.1, preuve section 6.4.

$$\Sigma \rightarrow_{id} \Sigma \cup \{x.r = x.p \mid x \in \mathcal{V}ars(\Sigma), \quad (p,r) \in I_\infty\}$$

est un opérateur d'évaluation partielle. La preuve est donnée dans la section 6.7 et ressemble un peu à la preuve du théorème de cohérence.

5.7 Arithmétique

Je signale dans cette section une petite transformation qui semble presque exotique dans le contexte des transformations précédentes. En général, la relation de réécriture ne concerne guère que des cas du genre $(+\ 2\ 3) \triangleright 5$. Mais on peut aussi enrichir considérablement cette relation, en lui ajoutant par exemple les règles suivantes:

$$\begin{array}{lcl}
(+\ a\ 0) & \triangleright & a \\
(+\ a\ b) & \triangleright & (+\ b\ a) \\
(+\ (+\ a\ b)\ c) & \triangleleft\triangleright & (+\ a\ (+\ b\ c)) \\
(+\ (*\ a\ b)\ (*\ a\ c)) & \triangleright & (*\ a\ (+\ b\ c)) \\
\ldots & &
\end{array}$$

la notation $p\triangleleft\triangleright q$ étant simplement une abréviation des deux règles $p\triangleright q$ et $q\triangleright p$. Évidemment, cette transformation fait très vite grossir la taille des systèmes, mais elle peut aussi être une source importante de simplifications dans les programmes qui font des calculs, notamment pour propager des simplifications.

5.8 Réduction de Programme

Cette transformation est importante sur le plan pratique car, à l'inverse des précédentes, elle permet de réduire la taille d'un programme. Son défaut: elle n'est correcte que pour certaines formes de dérivation.

L'idée de la transformation est simple. Si on calcule la sémantique équationnelle du programme $\mathcal{P}rg$ (Cf. chapitre 8), on a un ensemble de stratégies qui permettent de calculer des dérivations pour un certain champs d'observation $\mathcal{O}bs$.

Ces stratégies sont formées de séquences de pas élémentaires qui sont à leur tour formées d'appels récursifs de stratégies et d'application d'équations du programme.

Si on récolte les équations qui sont présentes dans au moins une stratégie, on obtient un nouveau programme $\mathcal{P}rg'$ qui est, par définition, équivalent à $\mathcal{P}rg$ mais uniquement pour les dérivations saturées.

Telle est l'idée de base de la réduction de programme. Mais je peux aller plus loin encore. Pour un triplet (P, R, c), il peut exister plusieurs séquences seq telles que (P, R, c, seq) soit une stratégie du programme $\mathcal{P}rg$ (Je rappelle que (P, R, c, seq) est une stratégie si la séquence seq permet de calculer les attributs R à partir des attributs P dans le cas où $\alpha = (c \ldots)$).

Appliquer l'une ou l'autre des stratégies est équivalent pour les valeurs calculées, mais toutes les séquences ne sont pas aussi efficaces les unes que les autres: elles peuvent correspondre à des dérivations plus ou moins longues.

La réduction que je vais définir permet de choisir pour chaque triplet (P, R, c) une séquence qui soit *a priori* plus efficace que les autres. On récolte alors uniquement les équations impliquées dans ce choix pour former le programme $\mathcal{P}rg'$. On obtient alors un programme plus petit et plus efficace que $\mathcal{P}rg$.

5.8.1 La fonction de choix

Pour faire le choix des séquences à considérer, je vais utiliser une métrique un peu "*ad-hoc*", comme on dit! Je ne garantis pas l'optimalité du choix car c'est un problème difficile à définir et à calculer, et de toute façon certainement indécidable. Je ne me base donc que sur mon intuition.

Pour chaque séquence *seq*, je définis une métrique $M(seq)$ qui mesure l'efficacité intuitive de la séquence. En fait, j'en définis plusieurs, et je fais une moyenne pondérée des différentes mesures. Plus la mesure est haute, moins la séquence sera efficace. Voici la liste des métriques que j'utilise (on peut en utiliser d'autres):

- Longueur: la longueur de la séquence.
- Cohérence: nombre d'attributs autres que P et R présents dans la séquence. Cette métrique permet d'éviter certains entrelacement inutiles de fonctions mutuellement récursives.
- Composition: nombre d'appels récursifs sur des variables locales qui sont elles-mêmes des résultats d'appels récursifs. C'est à dire, nombre de pas élémentaires de la forme $(P', R', \alpha.L_k)$ tels qu'il existe un autre pas de la forme $\alpha.L_k = x.r$. Cette métrique sert à sélectionner des séquences qui font peu de composition, donc qui ont de forte chances de construire peu de structures intermédiaires.
- Évaluation partielle: nombre d'appels récursifs sur des variables locales dont on connaît le constructeur de tête. Cette métrique permet de favoriser les séquences partiellement évaluées.
- (etc.)

Les coefficients sont à ajuster en fonction du goût de chacun. Pour la métrique M, la fonction de choix C_M est alors la suivante:

$$C_M(P, R, c) = seq \;\left|\; \begin{array}{c} (P, R, c, seq) \in [\![\mathcal{P}rg]\!]^{op} \\ M(P, R, c, seq) \text{ minimum} \end{array}\right.$$

5.8.2 La transformation

Je commence tout d'abord par définir la fonction S_c qui extrait d'une séquence des équations de programme relatives au constructeur c:

$$S_c(seq) = \{c \to x = t \mid (x = t) \in seq\}$$

Alors la réduction du programme $\mathcal{P}rg$ est simplement définie par:

$$\mathcal{T}rans_{red}(\mathcal{P}rg) = \bigcup \{S_c(C_M(P,R,c)) \mid c \in \tau_R \quad (P,R) \in \mathcal{O}bs\}$$

C_M étant la fonction de choix définie plus haut et $\mathcal{O}bs$ étant le champs d'observation maximal associé au programme $\mathcal{P}rg$.

La transformation est correcte pour les dérivations saturées, c'est à dire que l'on a le théorème suivant:

$$[\![\mathcal{T}rans_{red}(\mathcal{P}rg)]\!]^{op} = [\![\mathcal{P}rg]\!]^{op}$$

5.9 Implémentation

Cette section regroupe des problèmes d'implémentation pour les transformations considérées.

5.9.1 Spécialisation: cas général

Un problème de redondance peu très vite survenir quand on fait plusieurs spécialisations de la même fonction. En effet, prenons deux valeurs K_1 et K_2 et une fonction f de trois variables. Je définis les fonctions suivantes:

$$\begin{array}{l}
f_1(x,z) = f(x,K_1,z) \\
f_2(x,y) = f(x,y,K_2) \\
f_3(x) = f_1(x,K_2) \\
f_4(x) = f_2(x,K_1)
\end{array}$$

Bien évidement, on a pour tout x la relation $f_3(x) = f_4(x) = f(x,K_1,K_2)$. Si on veut limiter une expansion exponentielle du nombre de d'attributs qui calculent tous les même valeurs, il faut donc garder une trace des spécialisations déjà effectuées. Par rapport au cas de base, on va donc gérer globalement toutes les spécialisations dans une même transformation.

J'appelle contexte de spécialisation un ensemble de couples (p,K) avec $p \in \mathcal{P}rm$ et K une valeur. J'appelle conséquence de spécialisation un ensemble de couples (r,s) avec $r \in \mathcal{R}es \cap \mathcal{O}$ et $s \in \mathcal{R}es - \mathcal{O}$, où $\mathcal{O}$ est l'ensemble des attributs du champs d'observation du programme initial.

Soit SPE une table qui donne pour tout contexte (de spécialisation) la conséquence associée (de spécialisation). Pour chaque entrée (C,S) dans cette table, je définis un programme additionnel $\mathcal{P}rg'_{(C,S)}$ et une relation $D_{(C,S)}$ suivante. Je note P l'ensemble des attributs paramètres présents dans le contexte C. Je note R l'ensembles des attributs résultats r tels que $(r,s) \in S$. Le programme $\mathcal{P}rg'$ est identique pour tous les constructeurs $c \in \tau_R$ (voir section 3.5.6).

$$\mathcal{P}rg'_{C,S} = \bigcup_{c \in \tau_R} \left\{ \begin{array}{llll}
c & \rightarrow & \alpha.s = \alpha.L_m.r & (r,s) \in S \\
c & \rightarrow & \alpha.L_m.p = K & (p,K) \in C \\
c & \rightarrow & \alpha.L_m.p = \alpha.p & p \in \mathcal{D}ep(R) - P
\end{array} \right\}$$

$$\Sigma_1 D_{C,S} \Sigma_2 \Leftrightarrow (\exists x) \left\{ \begin{array}{l} (\forall (p,K) \in C) \quad x.p = K \in \Sigma_1 \\ \Sigma_2 = \Sigma_1 \cup \{x.r = x.s \mid (r,s) \in S\} \end{array} \right.$$

Les relations $\rightarrow_{\mathcal{P}rg'_{C,S}}$ et $D_{C,S}$ sont des opérateurs d'évaluation partielle. Il ne reste plus qu'à gérer efficacement les tables de composition.

Mise en œuvre: étant donné un attribut p et une valeur K, on peut ajouter à la table SPE la spécialisation correspondante par l'algorithme suivant:

$AddSpe(p, K)$	ajoute une spécialisation quand p vaut K:
1	rechercher $R = \{r \in \mathcal{R}es \cap \mathcal{O}bs \mid p \in \mathcal{D}ep(r)\}$.
2	ajouter dans la table SPE une association entre $\{(p, K)\}$ et $\{(r, newatt()) \mid r \in R\}$.
3	pour toute entrée (C, S) dans la table telle que p n'est pas dans C, calculer $S' = \{(r, newatt()) \mid (r, \_) \in S\}$ et ajouter dans la table SPE une association entre $C \cup \{(p, K)\}$ et S'.

Les démons décris au chapitre 7 utilisent cette table de spécialisation pour effectuer directement les transformations.

5.9.2 Composition: cas général

En pratique, de même que pour la spécialisation, il faut faire attention aux multiples compositions qui peuvent interagir les unes entre les autres. Si on ne fait pas attention, on risque de calculer de nombreuses compositions qui sont égales – par associativité de la composition des fonctions – et qui vont donc calculer exactement la même chose.

Prenons un exemple avec deux compositions. Pour simplifier, imaginons donc que nous ayons trois paires (p_1, r_1), (p_2, r_2) et (p_3, r_3) dans le champs d'observation du programme initial.

Si on cherche a composer r_1 et r_2, on aura un nouvel attribut résultat c_1 et un nouvel attribut paramètre q_1 qui servira à transmettre le paramètre p_2. Ce que je note ainsi:

$$\begin{array}{l} r_1 \circ r_2 = c_1 \\ r_1 \circ p_2 = q_1 \end{array}$$

Pour utiliser cette table dans une composition, il suffit de procéder ainsi. Pour composer r_1 et r_2, je lis dans la table qu'il faut calculer c_1. Pour avoir le nom du renommage de p_2 dans la composition sur r_1, je cherche à quoi est associé $r_1 \circ p_2$, et je trouve q_1. Si je continue dans la même veine, on aura:

$$\begin{array}{l} r_2 \circ r_3 = c_2 \\ r_2 \circ p_3 = q_2 \end{array}$$

A noter que l'on a les dépendances (à priori) suivantes:

$$\begin{array}{l} \mathcal{D}ep(c_1) \subset \{p_1,\ q_1\} \\ \mathcal{D}ep(c_2) \subset \{p_2,\ q_2\} \end{array}$$

Si maintenant on compose c_1 et r_3, on aura donc:

$$\begin{array}{l} c_1 \circ r_3 = c_3 \\ c_1 \circ p_3 = q_3 \\ \mathcal{D}ep(c_3) \subset \{p_1,\ q_1,\ q_3\} \end{array}$$

Un problème survient si maintenant on veut composer r_1 et c_2. Normalement, on doit retrouver c_3, car c'est la même composition, mais associée différemment. Donc on devrait avoir la table suivante:

$$\begin{array}{l} r_1 \circ c_2 = c_3 \\ r_1 \circ p_2 = q_1 \\ r_1 \circ q_2 = q_3 \end{array}$$

Dans les tables que je viens d'écrire, on a une certaine cohérence quand on remplace par exemple c_2 et c_1 par leur origine, a savoir que:

$$\begin{array}{lll} r_1 \circ c_2 & = r_1 \circ (r_2 \circ r_3) & = c_3 \\ c_1 \circ r_1 & = (r_1 \circ r_2) \circ r_3 & = c_3 \end{array}$$

On retrouve donc la propriété associative de la composition, preuve que la représentation est intéressante. Plus généralement, on remarquera alors que tout peu être résumé dans la table suivante:

$$\begin{array}{rl} r_1 \circ r_2 & = c_1 \\ r_1 \circ p_2 & = q_1 \\ r_2 \circ r_3 & = c_2 \\ r_2 \circ p_3 & = q_2 \\ r_1 \circ r_2 \circ r_3 & = c_3 \\ r_1 \circ r_2 \circ p_3 & = q_2 \end{array}$$

Mise en œuvre: une fois admis le fait que c'est bien cette table qu'il faut maintenir à jour, il reste trois algorithmes à préciser. Le premier concerne l'insertion d'une nouvelle composition, et les deux suivants concernent respectivement de deux opérateurs d'évaluation partielle. Un autre algorithme utile sera celui qui retrouve l'origine d'une composition dans la table (l'image réciproque de la table en quelque sorte) et celui qui ajoute une nouvelle entrée si elle ne s'y trouve pas déjà.

J'appelle donc *COMP* une table qui associe une séquence d'attributs avec un nouvel attribut. Les algorithmes voulus sont alors décris ci-dessous. Je commence par les deux outils de base:

FindOrigine a	trouve l'origine de a dans la table:
•	si il existe une séquence d'attributs u telle que $COMP[u] = a$ alors renvoyer u,
•	sinon renvoyer la séquence réduite à un élément $[a]$.
GetNew u	ajoute et renvoie la nouvelle entrée pour la liste u:
•	si $COMP[u] = a$, renvoyer a.
•	sinon choisir un nouveau nom d'attribut a et l'insérer dans la table avec $COMP[u]{:=}a$. Puis renvoyer a.

Alors, l'insertion d'une composition entre les attributs r et s s'effectue au moment où on la veut, comme suit. La concaténation de deux séquences u et v est notée $(u;v)$.

GetCompo a b	ajoute et renvoie une composition entre a et b dans la table $COMP$.
1	calculer $u = FindOrigine\ a$ et $v = FindOrigine\ b$,
2	renvoyer $GetNew\ (u;v)$.

Le premier opérateur d'évaluation partielle est simplement de la forme $\rightarrow_{\mathcal{P}rg'}$, où $\mathcal{P}rg'$ est un programme dont toutes les équations sont hors-champs. Le programme additionnel correspondant à la composition de r et s est le suivant:

$$\mathcal{P}rg'_{r,s} = (\forall c \in \tau_r) \left\{ \begin{array}{lll} c & \rightarrow & \alpha.a = \alpha.L_M.s \\ c & \rightarrow & \alpha.L_M = \alpha.r \\ c & \rightarrow & \alpha.L_M.p = \alpha.q_{(p,r)} \quad (p \in \mathcal{D}ep(r)) \end{array} \right\}$$

Avec L_M une nouvelle variable locale, $a = GetCompo\ r\ s$, et p et q tels que pour tout p dans $\mathcal{D}ep(s)$, $q_{(p,r)} = GetCompo\ r\ p$. Suivant le même schéma, et les mêmes relations, le deuxième opérateur d'évaluation partielle est une relation entre systèmes définie par :

$$\Sigma_1\ D_{r,s}\ \Sigma_2 \Leftrightarrow (\exists x, y) \left\{ \begin{array}{l} x = y.r \in \Sigma_2 \\ \Sigma_2 = \Sigma_1 \cup \left\{ \begin{array}{cclc} x.s & = & x.L_N.a & \\ x.L_N & = & y & \\ x.L_N.p & = & y.p & (p \in \mathcal{D}ep(r)) \\ y.q_{(p,r)} & = & x.p & (p \in \mathcal{D}ep(r)) \end{array} \right\} \end{array} \right.$$

Encore une fois, L_N doit être une variable local fraîche à chaque utilisation de $D_{r,s}$. Le démon associé à cette extension de dérivation est décris au chapitre 7.

5.9.3 Précalculation: cas général

Le lecteur aura remarqué que l'on peut utiliser la même table que pour la transformation de composition afin de gérer les nouveaux noms d'attributs. Il reste alors à contrôler que pour un constructeur c donné, chaque composition n'est effectuée qu'une seule fois. J'utilise donc une autre table qui indique pour chaque constructeur c quels sont les attributs qui ont été précalculés. Cette table est notée *PreCalc*.

Comme pour la composition, á chaque entrée dans la table de composition correspond deux opérateurs d'évaluation partielle. Le premier est de la forme $\rightarrow_{\mathcal{P}rg'}$ où $\mathcal{P}rg'$ est un nouveau programme. Pour une composition des attributs p et r, ce programme est simplement :

$$\begin{array}{l} \mathcal{P}rg'_{p,r} = (\forall (c,x) \mid (\exists t)\ (c \rightarrow x.p = t) \in \mathcal{P}rg) \\ \left\{ \begin{array}{lll} c & \rightarrow & x.p' = \alpha.L_M.r \\ " & \rightarrow & \alpha.L_M = x.p \\ " & \rightarrow & \alpha.L_M.q = x.r_{(p,q)} \quad (\forall q \in \mathcal{D}ep(r)) \end{array} \right\} \end{array}$$

Avec bien entendu L_M qui est une nouvelle variable locale jamais utilisée, $p' = \mathit{GetCompo}\ p\ r$ et $r_{(p,q)} = \mathit{GetCompo}\ p\ q$ et pour tout $q \in \mathcal{D}ep(r)$. Le deuxième opérateur est une relation définie par :

$$\Sigma_1\ D_{p,r}\ \Sigma_2 \Leftrightarrow \left\{ \begin{array}{l} \alpha = (c \ldots) \in \Sigma_1 \\ x = \alpha.p \in \Sigma_1 \\ (p,r) \notin \mathit{PreCalc}\ c \\ \Sigma_2 = \Sigma_1 \cup \left\{ \begin{array}{ccl} x.r & = & \alpha.p' \\ \alpha.r_{(p,q)} & = & x.q \quad (\forall q \in \mathcal{D}ep(r)) \end{array} \right\} \end{array} \right.$$

Si $D_{r,s}$ est utilisée, alors on doit mettre à jour *PreCalc* en y ajoutant (p,r) pour le constructeur c correspondant.

Chapitre 6

Preuves de Correction

Ce chapitre contient les preuves des théorèmes essentiels aux transformations de programme. Il contient aussi la définition d'une relation de dépendance approchée qui me sert de base pour certaines récurrences. Ces preuves sont assez techniques, mais il n'est pas nécessaire de les comprendre pour pouvoir inventer de nouvelles transformations de programme: il suffit d'utiliser les théorèmes essentiels ou de faire des récurrences.

6.1 Les théorèmes

Dans le chapitre 5, j'ai pu présenter les quelques théorèmes utiles aux preuves de corrections. J'en donne ici les énoncés précis avec quelques commentaires.

Le point de départ est toujours en rapport avec l'évaluation partielle. Les définitions sont données à la section 5.1, et le théorème qui m'intéresse ici est le suivant:

Théorème 6.1.1 (Correction de l'évaluation partielle)

$$\mathcal{T}rans_{(EP)}(\mathcal{P}rg) \sim_{\mathcal{O}bs} \mathcal{P}rg$$

En plus de ce théorème, j'utilise essentiellement trois principes pour effectuer des transformations plus complexes. Le premier permet d'ajouter des programmes additionnels pourvu qu'ils portent sur des variables locales non-encore utilisées:

Théorème 6.1.2 (Théorème des Ajouts Inutiles)

Soit x_0 une variable hors-champs. Soit alors $\mathcal{X}$ l'ensemble des variables qui ont x_0 comme préfixe:

$$\mathcal{X} = \{x[x_0] \mid x \in \mathcal{V}ars\}$$

Alors, pour tout terme t_0, tout système Σ tel que $\mathcal{X} \cap \mathcal{V}ars(\Sigma) = \emptyset$ et tout système final $\Sigma' \in \mathcal{F}inal(\mathcal{O}bs)$, on a:

$$\Sigma \cup \{x_0 = t_0\} \Rightarrow_{\mathcal{P}rg} \Sigma' \quad \Rightarrow \quad \Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$$

Le second principe permet de repérer grâce à la notion de dépendance approchée des calculs d'attributs sur des variables différentes qui donneront des résultats identiques sous certaines conditions. C'est le théorème de cohérence dont l'énoncé est:

Théorème 6.1.3 (Théorème de Cohérence)
Soit un système Σ, deux variables x et y et un attribut résultat r. Alors on a:

$$\left\{ \begin{array}{ll} x =_\Sigma y & \\ x.p =_\Sigma y.p & \forall p \in \mathcal{D}ep(r) \end{array} \right\} \quad \Rightarrow \quad x.r =_\Sigma y.r$$

Enfin, le troisième principe permet d'ajouter une équation $x = y$ quand on sait que x et y ne peuvent recevoir que des valeurs identiques. Le théorème associé utilise la notion d'égalité de deux variables dans un contexte Σ, notion qui est définie à la section 5.2. Le théorème sous-jacent est:

Théorème 6.1.4 (Théorème d'extensions des égalités)
Soit la fonction E définie par:

$$(\forall \Sigma) \left\{ \begin{array}{ll} E(\Sigma) = \{x = y\} & \text{si} \quad (\exists x, y) \quad x =_\Sigma y \\ E(\Sigma) = \emptyset & \text{sinon.} \end{array} \right.$$

Alors E est un opérateur d'évaluation partielle.

Ce théorème permet en pratique de prouver les extensions de dérivations utilisées pour les transformations sont des extensions correctes.

6.2 Dépendances Approchées

Les dépendances approchées sont définies par un point fixe sur des fermetures de graphes. En effet, partant d'une dépendance *a priori* que je note D, on peut calculer une dépendance *a posteriori* pour chaque constructeur c, en considérant les dépendances locales induites par les équations du programme $\mathcal{P}rg$.

Plus précisément, considérons un constructeur c et une équation $c \to x = t$ du programme $\mathcal{P}rg$. Intuitivement, pour pouvoir calculer x avec cette équation, il faut au moins pouvoir calculer les variables y qui apparaissent dans t. Je définis alors le graphe $G_{loc}(c)$, qui représente les dépendances locales au constructeur c par:

$$G_{loc}(c) = \{\mathcal{V}ars(t) \to x \mid c \to x = t \in \mathcal{P}rg\}$$

Ce graphe ne tient pas compte des dépendances induites par les calculs récursifs sur les variables $\mathcal{X}_{loc}$ et je vais utiliser la relation de dépendance D pour les modéliser. Je définis donc le graphe des appels récursifs $G_{rec}(D)$ par:

$$G_{rec}(D) = \{A \to x.r \mid A = \{x\} \cup \{x.p \mid p \in D(r)\}, \quad x \in \mathcal{X}_{loc}\}$$

Remarque: Ces graphes sont représentés de manière particulière, les arcs étant regroupés par paquets. Ainsi, dans un graphe G, il y a un chemin entre x et y si il existe un ensemble A tel que $x \in A$ et $A \to y \in G$. Voir section 10.7. Ce regroupement est utile par la suite, pour les preuves.

Je considère maintenant le graphe total :

$$G(c, D) = G_{loc}(c) \cup G_{rec}(D)$$

Si dans ce graphe il existe une chemin entre $\alpha.p$ et $\alpha.r$, alors il y a de fortes chances que r dépende de p. Donc, on devrait avoir $p \in \mathcal{D}ep(r)$.

Pour trouver ces chemins, j'utilise la fermeture transitive de ce graphe:

$$F(c, D) = (G(c, D))^*$$

Voir section 10.7 pour le calcul de cette fermeture. On a alors un graphe qui contient toutes les dépendances possibles pour le constructeur c. En particulier, je peux définir l'ensemble des attributs paramètres dont dépend un attribut résultat r par:

$$\varphi(c, D)(r) = \{p \mid \alpha.p \to \alpha.r \in F(c, D)\}$$

Toutefois, cette dépendance n'est valable que pour le constructeur c. Pour obtenir *a posteriori* une nouvelle relation de dépendance, j'utiliserai donc la fonction ψ définie par:

$$\psi(D)(r) = \bigcup_c \varphi(c, D)(r)$$

Si D est un bon candidat pour la relation $\mathcal{D}ep$, on aura donc pour tout r, $\psi(D)(r) = D(r)$, c'est à dire, $\psi(D) = D$. Alors on aura la définition finale des dépendances approchées:

$$\mathcal{D}ep = \text{lfix } \psi$$

D'après la section 10.2, le lecteur pourra vérifier que si $D_\emptyset$ est la relation vide – telle que $D_\emptyset(r) = \emptyset$ pour tout r – alors on a:

$$\mathcal{D}ep = \bigcup_{m \geq 0} \psi^m(D_\emptyset)$$

Avec cette définition, on peut démontrer le théorème de cohérence. La preuve est technique mais suit la définition du graphe $F(c, \mathcal{D}ep)$. Le point fixe permet d'obtenir une récurrence un peu étrange fondée sur la structure des variables rencontrées au cours des dérivations. Ce théorème n'est pas facile à démontrer, mais il donne un résultat très riche qui permet de simplifier énormément les preuves des transformations de programmes. C'est en pratique le théorème le plus important pour ces preuves et c'est pourquoi je présente sa démonstration au chapitre 6.

Une autre utilisation de cette définition de $\mathcal{D}ep$ permet de démontrer le théorème de décomposition. Ce théorème est utile dans deux cas: pour prouver l'existence de stratégies pour certaines formes de dérivations (Cf. chapitre 8) et pour la preuve de certaines transformations de programmes.

6.3 Évaluation Partielle

Rappelons l'énoncé du théorème à démontrer:

Théorème 6.3.1 (Correction de l'évaluation partielle)

$$\mathcal{T}rans_{(EP)}(\mathcal{P}rg) \sim_{\mathcal{O}bs} \mathcal{P}rg$$

Pour prouver le théorème, je me propose de démontrer que, pour tout système Σ et tout système $\Sigma' \in \mathcal{F}inal(\mathcal{O}bs)$ on a:

$$\begin{array}{ll} & \Sigma \Rightarrow_{\mathcal{P}rg \cup \mathcal{T}rans(\mathcal{P}rg)} \Sigma' \\ \Leftrightarrow & \Sigma \Rightarrow_{\mathcal{P}rg} \Sigma' \end{array}$$

Par monotonie, on peut dire que les dérivations ($\Rightarrow_{\mathcal{P}rg}$) sont incluses dans les dérivations ($\Rightarrow_{\mathcal{P}rg' \cup \mathcal{P}rg}$). Donc, pour prouver le théorème, il suffit de montrer que, dans les mêmes conditions:

$$\begin{array}{ll} & \Sigma \Rightarrow_{\mathcal{P}rg \cup \mathcal{T}rans(\mathcal{P}rg)} \Sigma' \\ \Rightarrow & \Sigma \Rightarrow_{\mathcal{P}rg} \Sigma' \end{array}$$

Pour réaliser la preuve, je vais simplement faire une récurrence sur la longueur des dérivations $\Rightarrow_{\mathcal{T}rans(\mathcal{P}rg) \cup \mathcal{P}rg}$. Pour alléger les notation, je vais noter $\mathcal{P}rg' = \mathcal{T}rans(\mathcal{P}rg) - \mathcal{P}rg$. Je vais donc prouver pour tout n la propriété suivante, notée H_n:

$$(\forall \Sigma)(\forall \Sigma' \in \mathcal{F}inal(\mathcal{O}bs)) \left\{ \begin{array}{ll} & \Sigma \rightarrow^n_{\mathcal{P}rg' \cup \mathcal{P}rg} \Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma' \\ \Rightarrow & \Sigma \Rightarrow_{\mathcal{P}rg} \Sigma' \end{array} \right.$$

Pour $n = 0$, les dérivations se réduisent à une inclusion et il est facile de démontrer H_0. Supposons maintenant H_n, et considérons une dérivation de la forme:

$$\Sigma \rightarrow^n_{\mathcal{P}rg' \cup \mathcal{P}rg} \Sigma_1 \rightarrow_{\mathcal{P}rg' \cup \mathcal{P}rg} \Sigma_2 \Rightarrow_{\mathcal{P}rg} \Sigma'$$

Pour terminer la récurrence, il suffit alors de prouver que $\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma'$. On pourra alors appliquer H_n et ainsi prouver que $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$.

Je vais maintenant prouver $\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma'$. Or $\Sigma_1 \rightarrow_{\mathcal{P}rg' \cup \mathcal{P}rg} \Sigma_2$ donc deux cas se présentent. Ou bien $\Sigma_1 \rightarrow_{\mathcal{P}rg} \Sigma_2$, et dans ce cas on a bien $\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma'$. Ou bien, $\Sigma \rightarrow_{\mathcal{P}rg'} \Sigma_2$, et dans ce dernier cas, on a alors:

$$x = (c\ t_1 \dots t_n) \in \Sigma_1$$
$$c \to y = t \in \mathcal{P}rg'$$
$$\Sigma_2 = \Sigma_1 \cup \{y[x] = t[x]\}$$

On a donc, par substitution et propriété des sous termes la relation:

$$\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma_1 \cup \{x = (c\ x.1 \dots x.n)\}$$

La preuve du théorème revient donc à prouver le fait suivant:

$$\Sigma_1 \cup \{x = (c\ x.1 \dots x.n)\} \cup \{y[x] = t[x]\} \quad \Rightarrow_{\mathcal{P}rg} \Sigma'$$

Or $c \to y = t \in \mathcal{P}rg'$ correspond exactement à dire que:

$$\{\alpha = (c\ \alpha.1 \dots \alpha.n)\}\ (EP)^*\ y = t$$

Or, $(EP)^*$ étant un opérateur d'évaluation partielle, on obtient par application de ses propriétés de monotonie:

$$\Sigma_1 \cup \{x = (c\ x.1 \dots x.n)\}\ (EP)^*\ \Sigma_1 \cup \{x = (c\ x.1 \dots x.n)\} \cup \{y[x] = t[x]\}$$

Donc, en appliquant la propriété essentielle des opérateurs d'évaluation partielle, comme on sait que:

$$\Sigma_1 \cup \{x = (c\ x.1 \dots x.n)\} \cup \{y[x] = t[x]\} \Rightarrow_{\mathcal{P}rg} \Sigma'$$

alors on peut en déduire que:

$$\Sigma_1 \cup \{x = (c\ x.1 \dots x.n\} \Rightarrow_{\mathcal{P}rg} \Sigma'$$

donc:

$$\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma'$$

Ce qui termine la preuve.

6.4 Théorème d'extension des égalités

Rappelons le théorème:

Théorème 6.4.1 (Théorème d'extensions des égalités)
Soit la fonction E définie par:

$$(\forall \Sigma) \begin{cases} E(\Sigma) = \{x = y\} & \text{si} \quad (\exists x, y) \quad x =_\Sigma y \\ E(\Sigma) = \emptyset & \text{sinon.} \end{cases}$$

Alors E est un opérateur d'évaluation partielle.

La preuve est extrêmement simple. Il faut montrer que, pour tout système Σ' de $\mathcal{F}inal(\mathcal{O}bs)$, si on suppose $\Sigma \cup E(\Sigma) \Rightarrow_{\mathcal{P}rg} \Sigma'$, alors on a $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$. Prenons les deux cas de définition de la fonction E:

Cas 1: il existe x et y tels que $x =_\Sigma y$ et $E(\Sigma) = \Sigma \cup \{x = y\}$. On a donc par définition de $(=_\Sigma)$ la dérivation $\Sigma \Rightarrow_{\mathcal{P}rg} \{x = y\}$. Comme $\Rightarrow_{\mathcal{P}rg}$ est une relation déductive on a aussi $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma \cup E(\Sigma)$. Toujours en utilisant que $\Rightarrow_{\mathcal{P}rg}$ est déductive, on a bien par transitivité que $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$.

Cas 2: on a immédiatement $\Sigma \cup E(\Sigma) = \Sigma$, donc $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$.

6.5 Théorème des ajouts inutiles

Rappelons l'énoncé du théorème:

Théorème 6.5.1 (Théorème des Ajouts Inutiles)

Soit x_0 une variable hors-champs. Soit alors $\mathcal{X}$ l'ensemble des variables qui ont x_0 comme préfixe:

$$\mathcal{X} = \{x[x_0] \mid x \in \mathcal{V}ars\}$$

Alors, pour tout terme t_0, tout système Σ tel que $\mathcal{X} \cap \mathcal{V}ars(\Sigma) = \emptyset$ et tout système final $\Sigma' \in \mathcal{F}inal(\mathcal{O}bs)$, on a:

$$\Sigma \cup \{x_0 = t_0\} \Rightarrow_{\mathcal{P}rg} \Sigma' \quad \Rightarrow \quad \Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$$

La preuve est laborieuse à cause de nombreux cas à prendre en compte. Cependant, l'idée générale est plutôt simple. Elle consiste à partir d'une dérivation $\Sigma \cup \{x_0 = t_0\} \Rightarrow_{\mathcal{P}rg} \Sigma'$ et à construire une autre dérivation $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$. Pour construire cette dernière dérivation, je vais simplement couper tout ce qui se déduit de l'équation $x_0 = t_0$. Pourquoi? La propriété d'être hors-champs garanti que la variable x_0 n'interfèrera pas avec le système final Σ'.

Mais les substitutions vont poser des problèmes, car en effet, la variable x_0 peut subitement apparaître dans le reste de la dérivation, si par malheur t_0 est présent dans une dérivation à partir de Σ. Mais on peut s'affranchir du problème, en "saturant" les équations observées: je vais donc substituer autant que faire se peut toutes les occurrences des variables (ici $\mathcal{X}$) qui ont x_0 comme préfixe.

6.5.1 Définitions préliminaires

Définition: je note $\overline{\Sigma}$ le système constitué de toutes les équations de la forme $z = t$ avec $z \in \mathcal{X}$ et qui peuvent être déduites à partir de Σ :

$$\overline{\Sigma} = \{z = t \mid \Sigma \cup \{x_0 = t_0\} \Rightarrow_{\mathcal{P}rg} z = t, \quad z \in \mathcal{X}\}$$

Ce système $\overline{\Sigma}$ sera très utile pour s'affranchir des problèmes du substitutions avec les variables de l'ensemble $\mathcal{X}$. On a bien entendu $\Sigma \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}$ et $x_0 = t_0 \in \overline{\Sigma}$, donc on a aussi:

$$\Sigma \cup \overline{\Sigma} \Rightarrow_{\mathcal{P}rg} \Sigma'$$

Définition: Par définition de $\Rightarrow_{\mathcal{P}rg}$, on sait alors qu'il existe une suite $(\Sigma_i)_{i\leq n}$ telle que:

- $\Sigma_0 = \Sigma \cup \overline{\Sigma}$
- $(\forall i < n) \quad \Sigma_i \rightarrow_{\mathcal{P}rg} \Sigma_{i+1}$
- $\Sigma_n \supset \Sigma'$

Comme on a $\Sigma' \in \mathcal{F}inal(\mathcal{O}bs)$, et comme y n'est pas une valeur, la dernière propriété s'écrit aussi $\Sigma_n \supset \Sigma'$.

Maintenant, je vais "saturer" et "couper" les systèmes Σ_i en effectuant toutes les substitutions qui sont possibles sur les variables $\mathcal{X}$ et en retirant toutes les équations qui portent sur un variable de l'ensemble $\mathcal{X}$. La relation $\triangleright_{\overline{\Sigma}}$ définit une substitution d'une variable de $\mathcal{X}$, donc la saturation correspond à la relation $(\triangleright_{\overline{\Sigma}})^\infty$. La coupure se fait par simple filtrage. D'où la définition suivante:

Définition: Pour tout i, je définis le système suivant:

$$\overline{\Sigma}_i = \{z = t' \mid z = t \in \Sigma_i, \quad t(\triangleright_{\overline{\Sigma}})^\infty t', \quad z \notin \mathcal{X}\}$$

6.5.2 Preuve

Avec ces définitions, la preuve du théorème tient uniquement au lemme suivant:

Lemme: Pour tout i, on a $\Sigma \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_i$

En effet, si ce lemme est vrai, alors on a en particulier $\Sigma \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_n$. Soit alors une équation quelconque $z = v \in \Sigma'$. Par définition, on a $z = v \in \Sigma_n$. Comme v est une valeur, on a aussi $\mathcal{V}ars(v) = \emptyset$ et donc $v(\triangleright_{\overline{\Sigma}})^\infty v$. Donc $z = v \in \overline{\Sigma}_n$. Donc $\Sigma' \subset \overline{\Sigma}_n$. Donc, $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$.

Il ne reste plus que la preuve du lemme.

6.5.3 Preuve du Lemme

Elle se fait par récurrence sur i, et par cas suivant la règle qui est utilisée pour définir $\Sigma_i \rightarrow_{\mathcal{P}rg} \Sigma_{i+1}$. Intuitivement, dans la plupart des cas, on montrera que $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_{i+1}$. Le seul cas difficile est celui de la substitution et des primitives. Je les traite donc en dernier.

Pour $i = 0$, comme $\mathcal{X} \cap \mathcal{V}ars(\Sigma) = \emptyset$, alors on a $\overline{\Sigma}_0 = \Sigma$. Donc $\Sigma \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_0$.

Maintenant, supposant le lemme vrai pour un certain i, je vais montrer par cas sur $\Sigma_i \rightarrow_{\mathcal{P}rg} \Sigma_{i+1}$ que le lemme est encore vrai pour $i + 1$.

Sous-terme: On a $\Sigma_{i+1} = \Sigma_i \cup \{z.k = t_k\}$ avec $z = (c\ t_1 \ldots t_n) \in \Sigma_i$.

Si $z.k \in \mathcal{X}$ alors $\overline{\Sigma}_{i+1} = \overline{\Sigma}_i$. Donc $\Sigma \Rightarrow \overline{\Sigma}_{i+1}$. Sinon, $z.k \notin \mathcal{X}$, alors on a:

$$\overline{\Sigma}_{i+1} = \overline{\Sigma}_i \cup \{z.k = t'_k \mid t_k(\triangleright_{\overline{\Sigma}})^\infty t'_k\}$$

Remarquons que, si $z \in \mathcal{X}$, alors $z.k \in \mathcal{X}$. Donc, ici, $z \notin \mathcal{X}$. Or, comme $z = (c\, t_1 \ldots t_n) \in \Sigma_i$, on a aussi pour tout t'_j tels que $t_j (\triangleright_{\overline{\Sigma}})^\infty t'_j$, $z = (c\, t'_1 \ldots t'_n) \in \overline{\Sigma}_i$. Donc $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \{z.k = t'_k\}$. En utilisant l'hypothèse de récurrence, on a donc par transitivité $\Sigma \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_{i+1}$.

Programme: On a $\Sigma_{i+1} = \Sigma_i \cup \{a[z] = b[z]\}$ avec $z = (c \ldots) \in \Sigma_i$ et $c \to a = b \in \mathcal{P}rg$. Encore une fois, si $a[z] \in \mathcal{X}$, alors $\overline{\Sigma}_i = \overline{\Sigma}_{i+1}$ et on a le lemme pour $i+1$. Sinon, $a[z] \notin \mathcal{X}$ donc on a:

$$\overline{\Sigma}_{i+1} = \overline{\Sigma}_i \cup \{a[z] = t \mid b[z] (\triangleright_{\overline{\Sigma}})^\infty t\}$$

Remarquons encore une fois que, si $z \in \mathcal{X}$ alors $a[z] \in \mathcal{X}$. Donc, ici, $z \notin \mathcal{X}$. On a alors $z = (c \ldots) \in \Sigma_i$ et même $z = (c \ldots) \in \overline{\Sigma}_i$. Donc on peut affirmer que $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \{a[z] = b[z]\}$. Observons maintenant la relation $b[z] (\triangleright_{\overline{\Sigma}})^\infty t$. On a donc, pour tout $u \in \mathcal{V}ars(b[z])$, $u = t' \in \overline{\Sigma}$ et $b[z] \triangleright_{u=t'} b' (\triangleright_{\overline{\Sigma}})^\infty t$. Ce qui peut aussi s'écrire $b[z] \triangleright_{u=t''} b'' (\triangleright_{\overline{\Sigma}})^\infty t$ avec $t (\triangleright_{\overline{\Sigma}})^\infty t''$. Or $u = t' \in \overline{\Sigma}$ donc $u = t' \in \Sigma_i$, donc $u = t'' \in \overline{\Sigma}_i$. En observant cela pour toutes les variables $u \in \mathcal{V}ars(b[z])$, on a bien la relation $b[z] \triangleright_{\overline{\Sigma}_i} t$.

Donc $\{a[z] = b[z]\} \cup \overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_{i+1}$. Comme on a déjà $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \{a[z] = b[z]\}$, on a donc aussi $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_{i+1}$.

Primitives: On a $\Sigma_{i+1} = \Sigma_i \cup \{z = t_2\}$ avec $z = t_1 \in \Sigma_i$ et $t_1 \triangleright t_2$. Encore une fois, si $z \in \mathcal{X}$, alors $\overline{\Sigma}_i = \overline{\Sigma}_{i+1}$, et sinon:

$$\overline{\Sigma}_{i+1} = \overline{\Sigma}_i \cup \{z = t'_2 \mid t_2 (\triangleright_{\overline{\Sigma}})^\infty t'_2\}$$

L'idée consiste à utiliser le fait que les relations de réécriture sont monotones avec les substitutions[1].

Or la relation de substitution $(\triangleright_{\overline{\Sigma}})^\infty$ qu'il y a entre t_2 et t'_2 peut aussi s'écrire sous la forme: $(\exists \sigma)\ t'_2 = \sigma(t_2)$. Alors, la relation $\triangleright$ étant une relation de réécriture, comme on a $t_1 \triangleright t_2$, on a aussi $\sigma(t_1) \triangleright \sigma(t_2)$.

Remarquons que l'on a par définition de σ la relation $t_1 (\triangleright_{\overline{\Sigma}})^* \sigma(t_1)$. Maintenant, pour tout t'_1 tel que $\sigma(t_1) (\triangleright_{\overline{\Sigma}})^\infty t'_1$, il existe une substitution σ' telle que $t'_1 = \sigma' \circ \sigma(t_1)$.

En utilisant à nouveau que $\triangleright$ est une relation de réécriture, on a donc $\sigma' \circ \sigma(t_1) \triangleright \sigma' \circ \sigma(t_2)$.

Remarquons que l'on a aussi $t'_2 (\triangleright_{\overline{\Sigma}})^* \sigma'(t'_2)$.

Or on a $t_2 (\triangleright_{\overline{\Sigma}})^\infty t'_2$, donc nécessairement, $t'_2 = \sigma'(t'_2)$. Donc $t'_1 \triangleright t'_2$.

Par ailleurs, $z = t'_1 \in \overline{\Sigma}_i$. Donc $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_{i+1}$.

Substitution: On a $\Sigma_{i+1} = \Sigma_i \cup \{z = t_2\}$, avec $z = t_1 \in \Sigma_i$ et $t_1 \triangleleft\triangleright_{\Sigma_i} t_2$. Encore une fois, si $z \in \mathcal{X}$, alors $\overline{\Sigma}_i = \overline{\Sigma}_{i+1}$, et sinon:

$$\overline{\Sigma}_{i+1} = \overline{\Sigma}_i \cup \{z = t'_2 \mid t_2 (\triangleright_{\overline{\Sigma}})^\infty t'_2\}$$

1. Cf. section 10.4.

Comme on a $t_1 \triangleleft\!\triangleright_{\Sigma_i} t_2$ alors il existe nécessairement $u = v \in \Sigma_i$ tel que $t_1 \triangleleft\!\triangleright_{u=v} t_2$. Deux cas peuvent alors se présenter. Ou bien $u = v \in \Sigma_i$, ou bien $u = v$ est l'équation $x = y$. La preuve est différente dans les deux cas, je vais donc les aborder séparément. Mais dans les deux cas, je vais montrer que, pour tout t'_2 tel que $t_2(\triangleright_{\overline{\Sigma}})^\infty t'_2$ alors il existe t'_1 tel que $t_1(\triangleright_{\overline{\Sigma}})^\infty t'_1$ et $t'_1(\triangleleft\!\triangleright_{\overline{\Sigma}_i})^* t'_2$. On aura alors $z = t'_1 \in \overline{\Sigma}_i$ et donc $\overline{\Sigma}_i \Rightarrow_{\mathcal{P}rg} \overline{\Sigma}_{i+1}$.

- Cas où $u = v \in \Sigma_i$. Prenons la définition de $t_1 \triangleleft\!\triangleright_{u=v} t_2$: si $t_1 = u$ et $t_2 = v$, alors $t_2(\triangleright_{\overline{\Sigma}})^\infty t'_2$ reviens à $u(\triangleright_{\overline{\Sigma}})^\infty t'_2$, donc $t'_1 = t'_2$ convient; idem si $t_2 = u$ et $t_1 = v$; sinon, on propage cette propriété par récurrence sur la hauteur de t_1 (ou t_2) et donc on peut construire t'_1 tel que $t'_1(\triangleleft\!\triangleright_{\overline{\Sigma}_i})^* t'_2$.
- Cas où $u = v$ est l'équation $x = y$. De même, on prends la définition de $t_1 \triangleleft\!\triangleright_{x=y} t_2$: si $t_1 = x$ et $t_2 = y$, alors $t_2(\triangleright_{\overline{\Sigma}})^\infty t'_2$ signifie tout simplement $y = t'_2 \in \overline{\Sigma}$. Or $x =_\Sigma y$ par hypothèse, donc nécessairement $x = t'_2 \in \overline{\Sigma}$. Donc $t_1(\triangleright_{\overline{\Sigma}})^\infty t'_2$. Donc encore une fois, $t'_1 = t'_2$ convient. De même si $t_1 = y$ et $t_2 = x$.

Voila, cette preuve laborieuse est terminée.

6.6 Théorème de cohérence

Rappelons l'énoncé du théorème, qui est associé directement à la notion de dépendance approchée décrite à la section 3.5.6:

Théorème 6.6.1 (Théorème de Cohérence)
Soit un système Σ, deux variables x et y et un attribut résultat r. Alors on a:

$$\left\{ \begin{array}{ll} x =_\Sigma y & \\ x.p =_\Sigma y.p & \forall p \in \mathcal{D}ep(r) \end{array} \right\} \quad \Rightarrow \quad x.r =_\Sigma y.r$$

6.6.1 Idée générale

Si je résume le théorème, il suffit – par symétrie de x et y – de démontrer que si $\Sigma \Rightarrow_{\mathcal{P}rg} x.r = t$ alors $\Sigma \Rightarrow_{\mathcal{P}rg} y.r = t$.

La preuve du théorème est un peu longue, mais chaque étape est assez simple. Tout d'abord je voudrais faire la remarque suivante: seule l'application d'une équation de programme permet d'introduire une variable de la forme $x.r$ (voir la liste des quatre règles de dérivation, section 3.5).

Donc fatalement, on a $\Sigma \Rightarrow_{\mathcal{P}rg} x = (c \ldots)$. Comme $x =_\Sigma y$, on a aussi $\Sigma \Rightarrow_{\mathcal{P}rg} y = (c \ldots)$. L'idée de la preuve est alors simple: pour produire une équation de la forme $x.r = t$ on a nécessairement appliqué des équations du programme qui sont associées au constructeur c. On peut donc appliquer les mêmes équations pour y, et c'est pourquoi on aura les mêmes termes pour $x.r$ et $y.r$.

6.6.2 Variables atteignables par $\mathcal{D}ep$

Je reprends les définitions utilisées pour le calcul de la relation de dépendance approchée $\mathcal{D}ep$ (Cf. section 6.2). Considérons alors le graphe $G_c = G_{loc}(c) \cup G_{rec}(\mathcal{D}ep)$. A partir d'un ensemble de variables X, je peux définir les variables atteignables à partir de X dans G par:

$$A_c(X) = X \cup \{y \mid \exists (A \to y) \in G_c) \quad A \subset X\}$$

Soit alors $P_r = \{\alpha.p \mid p \in \mathcal{D}ep(r)\}$, on a par définition de $\mathcal{D}ep$ la propriété $\alpha.r \in A_c^*(P_r)$. Donc il existe un entier n tel que $\alpha.r \in A_c^n(P_r)$. L'idée principale de la démonstration du théorème consiste alors à démontrer pour tout n et pour toute variable $u \in A_c^n(P)$ que l'on a $u[x] =_\Sigma u[y]$.

Cependant, du fait que G contient des appels récursifs sur les variables $\mathcal{X}_{loc}$, il faudra mettre en place une récurrence un peu spéciale selon la "profondeur" des variables considérées. Je commence donc par expliquer cette notion.

6.6.3 Profondeur des variables

Prenons par exemple la variable $\alpha.L_4.5$. Je peux construire cette variable "couche par couche": je pars de α, je lui ajoute "$.L_4$" pour la première couche, et je lui ajoute ".5" pour la seconde couche. J'obtiens ainsi une variable de profondeur 2, car il me faut deux couches. Plus généralement, je peux construire une variable structurelle par couches, on comptant le nombre de fois qu'il faut ajouter un "$.k$" ou un "$.L_k$" à la variable "racine" α.

Je me penche maintenant sur un ensemble de variables particulières, que je note $\mathcal{X}_{loc}$ et que je définis comme suit:

Définition 6.6.1 (Ensemble $\mathcal{X}_{loc}$)
$\mathcal{X}_{loc}$ est l'ensemble des variables locales et sous-termes utilisées dans le programme $\mathcal{P}rg$, directement ou par le biais d'un attribut.

Ainsi, soit φ la fonction telle que $\varphi(x) = y$ si il existe un attribut a tel que $x = y.a$, et telle que $\varphi(x) = x$ dans le cas contraire. Alors:

$$\mathcal{X}_{loc} = \{\varphi(x) \mid x \in \mathcal{V}ars(\mathcal{P}rg)\}$$

Pour revenir à la notion de "couche", ajouter un suffixe "$.k$" ou "$.L_k$" à une variable x consiste aussi à instancier une variable de $\mathcal{X}_{loc}$ sur x. Si je note φ_{loc} la fonction qui ajoute une couche à tout un ensemble de variables, cette fonction peut s'écrire:

$$\varphi_{loc}(\mathcal{X}) = \{u[x] \mid u \in \mathcal{X}_{loc} \quad x \in \mathcal{X}\}$$

Définition 6.6.2 (Ensembles $\mathcal{X}_m$)
La couche des variables de profondeur n est l'ensemble ${\varphi_{loc}}^n(\{\alpha\})$.

Par ailleurs, je définis $\mathcal{X}_m$, ensemble des variables de profondeur supérieure à m par:

$$\mathcal{X}_m = \bigcup_{k \geq m} {\varphi_{loc}}^k(\alpha)$$

Notons que les ensembles $\mathcal{X}_m$ sont stables par φ_{loc}, et que plus précisément, pour tout $x \in \mathcal{X}_m$ on a $\varphi_{loc}(x) \in \mathcal{X}_{m+1}$.

6.6.4 Preuve du théorème

Je rappelle le contexte de la preuve. Soit Σ un système quelconque fixé. Je vais maintenant fixer également x, y et r. Je cherche à prouver le fait suivant:

$$\begin{array}{l} (\forall n)(\forall u \in A_c^n(P_r)) \\ \quad \left(\begin{array}{l} x =_\Sigma y \\ x.p =_\Sigma y.p \quad \forall p \in \mathcal{D}ep(r) \end{array} \right) \Rightarrow u[x] =_\Sigma u[y] \end{array}$$

avec $P_r = \{\alpha.p \mid p \in \mathcal{D}ep(r)\}$ et $c \in \tau_r$. Pour mettre en place la récurrence, il me faut un petit artifice pour limiter la profondeur des autres variables à considérer. C'est un détour important pour terminer la récurrence qui me permet de prouver le théorème.

Dans le cadre d'une preuve de transformation de programme, on devra utiliser le théorème de cohérence pour un système Σ tel que $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$ avec $\Sigma' \in \mathcal{F}inal(\mathcal{O}bs)$. Par définition, on a donc $\Sigma \rightarrow^n_{\mathcal{P}rg} \Sigma''$ pour un certain n avec $\Sigma' \subset \Sigma''$. Dans cette dérivation, toutes les variables qui sont susceptibles d'intervenir dans une preuve sont donc en nombre *fini*.

Définition: Je note $\mathcal{Y}$ cet ensemble de variables.

Avec cette limitation, je vais donc démontrer le théorème en prouvant que la relation $T(m,n)$ définie ci-dessous est vérifiée pour tous m et n:

$$T(m,n) \Leftrightarrow \left\{ \begin{array}{l} (\forall x \in \mathcal{X}_m \cap \mathcal{Y}, \quad \forall y \in \mathcal{X}_m \cap \mathcal{Y}, \quad \forall r \in \mathcal{R}es) \\ (\forall u \in A_c^n(P_r)) \\ \quad \left(\begin{array}{l} x =_\Sigma y \\ x.p =_\Sigma y.p \quad \forall p \in \mathcal{D}ep(r) \end{array} \right) \Rightarrow u[x] =_\Sigma u[y] \end{array} \right.$$

avec toujours $P_r = \{\alpha.p \mid p \in \mathcal{D}ep(r)\}$ et $c \in \tau_r$. Par commodité, je note aussi:

$$T^+(m) \Leftrightarrow (\forall n) T(m,n)$$

Je vais procéder en deux temps. Tout d'abord, je vais prouver que l'on a $T^+(m+1) \Rightarrow T^+(m)$. Ce qui revient à dire que, si le théorème est vrai pour des couches assez profondes ($T^+(m+1)$), alors il est vrai pour des couches moins profondes ($T^+(m)$). Ensuite, je terminerai la démonstration en remarquant que l'on a pas besoin de considérer des variables infiniment profondes car $\mathcal{Y}$ est fini, ce qui me permettra d'achever la démonstration du théorème.

Pour le premier temps, j'utilise le lemme suivant, dont je donne la preuve dans la section suivante pour plus de lisibilité.

$$(\forall m) \quad T^+(m+1) \Rightarrow (\forall n) \quad T(m,n) \Rightarrow T(m,n+1) \quad (1)$$

En effet, comme par définition $A_c^0(P_r) = P_r$, on a immédiatement $(\forall m)T(m,0)$. Donc, on a *a fortiori*:

$$(\forall m) \quad T^+(m+1) \Rightarrow T(m,0) \quad (2)$$

En combinant (1) et (2) on obtient par récurrence sur n la relation:

$$(\forall m) \quad T^+(m+1) \Rightarrow (\forall n)T(m,n) \quad (3)$$

Ce qui est justement équivalent à $T^+(m+1) \Rightarrow T^+(m)$ pour tout m. Pour terminer la récurrence sur m, supposons qu'il existe un entier m_0 tel que $\neg T^+(m_0)$. Alors, comme par contraposition on a $\neg T^+(m) \Rightarrow \neg T^+(m+1)$, on aurait par récurrence $(\forall m \geq m_0)\neg T^+(m)$. Donc, on aurait entre autre:

$$(\forall m \geq m_0)(\exists x \in \mathcal{X}_m \cap \mathcal{Y})$$

Or $\mathcal{Y}$ est fini, donc il ne peut pas contenir de variable de profondeur arbitrairement grande. Conclusion: m_0 n'existe pas. Donc:

$$(\forall m) \quad T^+(m)$$

6.6.5 Preuve du lemme

Supposons $T^+(m+1)$ et $T(m,n)$. Je dois montrer $T(m,n+1)$. Soit donc $u \in A_c^{n+1}(P)$, il faut que je montre $\Sigma \Rightarrow_{\mathcal{P}rg} u[x] = u[y]$. Par définition, $u \in A_c^{n+1}$ implique qu'il existe un ensemble de variables $A \subset A_c^n(P_r)$ tel que $A \to u \in G_c$. Or, par définition de G_c (Cf. section 6.6.2), deux cas sont possibles:

Cas 1: $A = \mathcal{V}ars(t)$ et $c \to u = t$ est une équation du programme $\mathcal{P}rg$. Comme on sait que $x =_\Sigma y$ par hypothèse, on peut également appliquer cette équation de programme sur y. Donc on a:

$$\Sigma \Rightarrow_{\mathcal{P}rg} \left\{ \begin{array}{l} u[x] = t[x] \\ u[y] = t[y] \end{array} \right.$$

Or, pour tout $v \in \mathcal{V}ars(t)$, $v \in A$ donc $v \in A_c^n(P_r)$. Par ailleurs, $v[x] \in \mathcal{X}_m \cap \mathcal{Y}$ et $v[y] \in \mathcal{X}_m \cap \mathcal{Y}$. Donc, on peut appliquer $T(m,n)$ qui est une hypothèse du lemme. Donc on a $v[x] =_\Sigma v[y]$ pour tout $v \in \mathcal{V}ars(t)$. On en déduit:

$$\Sigma \Rightarrow_{\mathcal{P}rg} \sigma \quad \text{avec :} \quad \sigma = \{v[x] = v[y] \mid v \in \mathcal{V}ars(t)\}$$

Or, il est facile de prouver que $t[x] \triangleright_\sigma t[y]$, donc on a naturellement par substitutions:

$$\Sigma \Rightarrow_{\mathcal{P}rg} \left\{ \begin{array}{l} u[x] = t[x] \\ u[y] = t[x] \end{array} \right.$$

D'où on tire finalement par une dernière substitution:

$$u[x] =_\Sigma u[y]$$

Cas 2: $u = z.r'$ pour certains $z \in \mathcal{X}_{loc}$ et $r' \in \mathcal{R}es$, et $A = \{z\} \cup \{z.p' \mid p' \in \mathcal{D}ep(r')\}$. Donc $u[x] = (z[x]).r'$. Mais on sait par ailleurs que $x \in \mathcal{X}_m \cap \mathcal{Y}$, or $z \in \mathcal{X}_{loc}$, donc nécessairement, $v[x] \in \mathcal{X}_{m+1} \cap \mathcal{Y}$ pour tout $v \in A$. De même pour y.

Comme on a $A \subset A_c^n(P_r)$, on peut donc affirmer que pour tout $v \in A$, on a $v[x] =_\Sigma v[y]$.

Si on récapitule ces résultats, on a donc $z[x] =_\Sigma z[y]$, pour tout $p' \in \mathcal{D}ep(r')$ on a $z[x].p' =_\Sigma z[y].p'$ et de plus $z[x] \in \mathcal{X}_{m+1} \cap \mathcal{Y}$ et $z[y] \in \mathcal{X}_{m+1} \cap \mathcal{Y}$. On peut donc appliquer $T^(m+1)$ qui est une hypothèse du lemme. Donc $z[x].r' =_\Sigma z[y].r'$. D'où on tire finalement:

$$u[x] =_\Sigma u[y]$$

Donc dans les deux cas, le lemme est démontré.

6.7 Élimination d'identité

Pour prouver que la relation $\rightarrow_{id}$ est un opérateur d'évaluation partielle, je vais en fait démontrer que pour tout $(p, r) \in I_\infty$ et pour tout x on a $x.r =_\Sigma x.p$.

En utilisant les mêmes remarques que pour le théorème de cohérence, je vais introduire la propriété $T(m, I)$ définie par:

$$T(m, I) \Leftrightarrow (\forall x \in \mathcal{X}_m \cap \mathcal{Y})(\forall (p, r) \in I) \quad x.r =_\Sigma x.p$$

Je vais alors prouver le lemme suivant:

$$T(m+1, I) \Rightarrow T(m, \Psi(I))$$

avec la fonction Ψ définie à la section 5.6. Cette fonction vérifie $I_\infty = \text{gfix } \Psi$. Donc par définition, on aura:

$$T(m+1, I_\infty) \Rightarrow T(m, I_\infty)$$

J'utilise alors la même observation que dans le théorème de cohérence sur le fait que $\mathcal{Y}$ est fini et que les profondeurs des variables à considérer sont bornées. Je peux alors finir la récurrence, et on a:

$$(\forall m)\ T(m, I_\infty)$$

Donc on a bien pour tout $(p, r) \in I_\infty$ et pour tout x on a $x.r =_\Sigma x.p$. On termine en utilisant le théorème d'extension des égalités pour prouver que $\rightarrow_{id}$ est bien un opérateur d'évaluation partielle.

Il reste à prouver le lemme. Supposons donc $T(m+1, I)$. Soit maintenant $x \in \mathcal{X}_m \cap \mathcal{Y}$ et $(p, r) \in \Psi(I)$. Pour tout constructeur c susceptible de définir x, on aura $c \in \tau_r$ et $\Sigma \Rightarrow_{\mathcal{P}rg} \{x = (c\ x.1 \ldots x.n)\}$. Donc par définition de Ψ, on aura:

$$\{\alpha = (c\ \alpha.1 \ldots \alpha.n)\} \cup \{y.r' = y.p' \mid (p', r') \in I, \quad y \in \mathcal{X}_{loc}\} \Rightarrow_{\mathcal{P}rg} \alpha.r = \alpha.p$$

Donc, on a en substituant le α par x:

$$\{x = (c\ x.1 \ldots x.n)\} \cup \{y[x].r' = y[x].p' \mid (p', r') \in I, y \in \mathcal{X}_{loc}\} \Rightarrow_{\mathcal{P}rg} x.r = x.p$$

Or, pour tout $y \in \mathcal{X}_{loc}$, $x \in \mathcal{X}_m$, on a $y[x] \in \mathcal{X}_{m+1}$. Donc pour tout $(p', r') \in I$, on peut appliquer $T(m+1, I)$ et obtenir:

$$\Sigma \Rightarrow_{\mathcal{P}rg} \{y[x].r' = y[x].p' \mid (p', r') \in I, y \in \mathcal{X}_{loc}\}$$

En combinant tout ces résultats on trouve finalement:

$$\Sigma \Rightarrow_{\mathcal{P}rg} x.r = x.p$$

Chapitre 7

Évaluation Partielle & Implémentation

Ce chapitre porte bien son nom! En effet, on a vu au chapitre 3 une définition de la sémantique équationnelle très théorique, mettant en jeu des notions difficilement calculables ou représentables, par exemple des relations, des ensembles, des limites, etc. Par exemple, est-il possible de calculer un système Σ' tel que $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$ pour un système Σ donné? Peut-on effectuer ce calcul en un temps raisonnable? Ma réponse sera franchement positive, mais il me faut avouer qu'il fallu franchir bien des obstacles pour y parvenir.

Dans un premier temps, je vais exposer les principaux problèmes qui se présentent et en cascade, les nouveaux problèmes posés par chaque embryon de solution. Pour clore cette introduction, il me suffit d'avertir le lecteur que la solution proposée ici pour le calcul effectif de $\Rightarrow_{\mathcal{P}rg}$ n'est pas la première que j'ai utilisée. J'essaierai en cours de chapitre d'esquisser d'autres solutions que j'ai implémentées mais qui restent à mon avis moins performantes.

Je signale au lecteur que les sections 7.1 et 7.2 contiennent l'essentiel des problèmes posés et ma façon de les résoudre. Comme toujours, les sections suivantes traiteront chacune d'un aspect de ces problèmes, en entrant dans les détails techniques et la description des algorithmes. J'invite donc le lecteur à ne les consulter qu'au fur et à mesure de ses besoins… et selon sa disponibilité!

7.1 Principaux Problèmes

7.1.1 Que faut-il calculer?

Rappelons brièvement le contexte. Nous avons un système d'équations Σ_0, des opérateurs d'évaluation partielle $\{OEP_i\}_{i \leq n}$ comprenant au moins $\rightarrow_{\mathcal{P}rg}$, et on cherche à calculer le système d'équations Σ_∞ défini par:

$$\Sigma_\infty = \bigcup \{\Sigma \mid \Sigma_0 (OEP_1 + \ldots + OEP_n)^* \Sigma\}$$

En effet, ce système regrouperait toutes les déductions possibles à partir des conditions initiales Σ_0. Signalons tout de suite que Σ_∞ a de bonnes chances d'être infini, ou du moins énorme, donc ce but est presque inaccessible en pratique. Ne nous démontons pas: nous ne sommes jamais intéressés par la totalité de Σ_∞. Mais, avant de savoir quel objet calculer, je propose d'examiner quelle utilisation on veut en faire.

Sur ce point, je ferai vite le tour de la question en la résumant ainsi: On veut les valeurs que peuvent prendre les variables d'un certain ensemble donné, que je note X. Autrement dit, on veut connaître tous les termes t tels que l'équation $x = t$ soit dans Σ_∞ avec $x \in X$.

En reprenant le sempiternel exemple du calcul de la longueur d'une liste, pour le système initial:

$$\Sigma_0 = \{\alpha = (\mathit{cons}\ 1\ (\mathit{cons}\ 2\ \mathit{nil}))\}$$

on est uniquement préoccupé par les valeurs de la variable $\alpha.\mathit{length}$. Mais même sur cet exemple simpliste, les problèmes pleuvent immédiatement.

7.1.2 Le problème des substitutions

Je voudrais exhiber le premier obstacle à franchir: celui des substitutions. Dans l'exemple précédant, les équations définissant la variable $\alpha.\mathit{length}$ dans Σ_∞ sont les suivantes:

$$\begin{array}{lcl} \alpha.\mathit{length} & = & (+\ 1\ \alpha.2.\mathit{length}) \\ \alpha.\mathit{length} & = & (+\ 1\ (+\ 1\ \alpha.2.2.\mathit{length})) \\ \alpha.\mathit{length} & = & (+\ 1\ (+\ 1\ 0)) \\ \alpha.\mathit{length} & = & (+\ 1\ 1) \\ \alpha.\mathit{length} & = & 2 \end{array}$$

sans oublier les équations qu'il faut ajouter pour définir les variables $\alpha.2.\mathit{length}$ et $\alpha.2.2.\mathit{length}$ qui ont subitement vue le jour:

$$\begin{array}{lcl} \alpha.2.\mathit{length} & = & (+\ 1\ \alpha.2.2.\mathit{length}) \\ \alpha.2.\mathit{length} & = & (+\ 1\ 0) \\ \alpha.2.\mathit{length} & = & 1 \\ \alpha.2.2.\mathit{length} & = & 0 \end{array}$$

On se rend compte ici de la complexité purement combinatoire introduite par les substitutions. En effet, si trente six équations définissent une variable x, chaque équation utilisant x en partie droite en induira trente six autres par substitutions. La solution à ce problème sera détaillée dans la section 7.3. La technique utilisée permettra de représenter ces équations modulo les classes d'équivalence introduites par la relation de substitution.

Pour l'exemple considéré, les classes d'équivalence obtenue sont les suivantes:

$$\begin{array}{lcl} C_1 & = & \{\ \alpha.2.2.\mathit{length},\ 0\} \\ C_2 & = & \{\ (+\ C_1\ C_0),\ \alpha.2.\mathit{length},\ 1\ \} \\ C_3 & = & \{\ (+\ C_1\ C_1),\ \alpha.\mathit{length},\ 2\ \} \end{array}$$

où C_1, C_2 et C_3 représentent les différentes classes d'équivalence. Tous les éléments d'une même classe sont égaux, et toutes les sous-expressions communes à deux termes ne sont représentées qu'une seule et même fois par une classe d'équivalence. Ainsi, une expression comme $(+\ C_1\ C_2)$ sera en fait l'ensemble des termes $(+\ t_1\ t_2)$ avec $t_1 \in C_1$ et $t_2 \in C_2$.

7.1.3 Remarque relative à la relation de réécriture.

Rappelons que la relation $\triangleright$ permet de définir des opérations ou des propriétés sur les primitives. Dans le chapitre 3, nous avons vu que cette relation est une règle de réécriture. Dans la mesure où elle permet de remplacer un terme par un autre, on aura donc le même genre de problème que pour la substitution. De fait, la solution proposée avec les classes d'équivalence sera encore une fois bien utile.

Vu le thème du chapitre, il me faudrait expliquer ici comment on calcule avec cette relation. Or, la représentation des termes par un ensemble de classes d'équivalence induit une profonde modification des techniques classiques associées aux règles de réécriture. Je renvoie donc le lecteur intéressé par ce problème à la section 7.6.

7.1.4 Variables nécessaires

On a vu au début de ce chapitre que l'on devait se limiter au calcul des valeurs d'un ensemble fini de variables. Mais déjà, dans la section précédente, on a vu qu'il fallait compléter cet ensemble initial par les variables intermédiaires nécessaires à leur calcul. Dans cette section, je voudrais mieux cerner cette notion de variable nécessaire.

Voyons par exemple le programme d'inversion de liste. Je vais compliquer un peu la situation en supposant que le système initial est maintenant le suivant:

$$\Sigma_0 = \begin{cases} \alpha = (cons\ 1\ \alpha.2) \\ \alpha.rev\_h = nil \end{cases}$$

et que la variable dont les valeurs nous intéressent est maintenant $\alpha.rev$. La complication vient de la queue de la liste α, qui ici n'est pas définie: elle est donc représentée par la variable générique $\alpha.2$, sans aucune équation supplémentaire pour la définir. Si l'on s'en tient à une notion intuitive de nécessité, pour avoir la valeur de la variable $\alpha.rev$, on peut appliquer naïvement une équation du programme et obtenir:

$$\alpha.rev = \alpha.2.rev$$

Mais ce n'est pas suffisant! En effet, il y a dans le système Σ_∞ une autre équation très importante:

$$\alpha.2.rev\_h = (cons\ 1\ nil)$$

Pourquoi cette équation est-elle si importante? Tout simplement parce que nous savons que l'attribut *rev* dépend de l'attribut *rev_h* à cause des équations du programme. Plus explicitement, dans la mesure où toutes les stratégies de résolution d'une variable $x.rev$ utilisent la variable $x.rev\_h$, alors: si la première est jugée nécessaire, la seconde doit l'être également.

En pratique, je compléterait l'ensemble des variables nécessaires en le fermant par la relation de dépendance approchée $\mathcal{D}ep$.

7.2 Solution Générale

Nous avons au départ deux données: un système initial Σ_0 et un ensemble de variables X_0. On veut trouver les équations de Σ_∞ qui sont impliquées dans le calcul des variables X_0.

Je vais donc calculer deux suites, $(\Sigma_n)_{n\geq 0}$ et $(X_n)_{n\geq 0}$. La première suite représentera (sous forme de classes d'équivalence) le système d'équations recherché, et la seconde les variables nécessaires.

La résolution est dirigée par les variables nécessaires calculées dans X_n. Je veux dire par là que toutes les équations que l'on ajoutera à Σ_n pour former Σ_{n+1} ne concerneront que les variables de X_n. On aura bien entendu pour tout n la relation $\Sigma_0 \Rightarrow_{\mathcal{P}rg} \Sigma_n$, et pour calculer X_{n+1} en fonction de X_n, je poserai tout simplement:

$$X_{n+1} = X_n \cup \mathcal{V}ars(\Sigma_{n+1}) \cup \{y \in \mathcal{D}ep(x) \mid x \in X_n\}$$

Pour calculer Σ_{n+1} en fonction de Σ_n, ma première idée a été de considérer toutes les équations possibles telles que $\Sigma_n \rightarrow_{\mathcal{P}rg} x = t$, avec $x \in X_n$. Mais en pratique, beaucoup de calculs étaient inutilement refaits à chaque itération. Pour contrer ce problème, il y avait bien sûr beaucoup d'astuces possibles, que j'ai d'ailleurs expérimentées. Mais je trouvais que l'ensemble avait un côté très bricolage. Je propose donc une toute autre solution.

Prenons une variable $x \in X_n$. Imaginons que x soit de la forme $y.r$ où r est un attribut résultat *i.e.* $r \in \mathcal{R}es$. Alors tant que l'on a pas le constructeur de tête de la variable y, inutile d'essayer d'appliquer une équation du programme pour $y.r$. De même, si x est de la forme $z.2.p$, avec $p \in \mathcal{P}rm$, il faut attendre d'avoir le constructeur de tête de z, etc.

Mon idée est donc d'installer un mécanisme de *démons*. Un démon est un petit programme qui attend qu'il y ait suffisamment de renseignements dans Σ_n pour produire de nouvelles équations. A chaque nouvelle variable x, selon sa forme, je génère des démons qui se mettront en veille jusqu'au moment où ils pourront intervenir pour donner des équations sur la variable x. A chaque itération, je dois donc calculer les nouvelles variables et les démons associés, appliquer les démons qui peuvent l'être et laisser les autres en attente, enfin, gérer les nouvelles équations.

J'ai choisi de ne pas présenter dès maintenant tous ces détails de l'algorithme. Je préfère commencer par deux gros morceaux: les démons (section 7.4) et la représentation des systèmes (section 7.3). Je pourrai alors détailler le calcul du point fixe (section 7.5). Enfin, j'exposerai deux algorithmes importants mais fastidieux. Le premier (section 7.6) est l'algorithme d'unification revu et corrigé pour tenir compte de la représentation des termes par des classes d'équivalence. Cet algorithme est particulièrement important pour gérer convenablement l'application de la relation $\triangleright$ sur les primitives. Le deuxième algorithme (section 7.7) concerne la restitution des résultats. En effet, n'oublions pas que l'on cherche en définitive un système d'équations $\Sigma \subset \Sigma_\infty$. Il faudra donc pouvoir transformer un ensemble de classes en un système équivalent.

7.3 Classes d'Equivalences

Oublions momentanément le contexte sémantique. Au fond, j'ai un problème simple: je dois représenter un ensemble de termes en les regroupant selon des classes d'équivalence d'une relation R. De plus, R contient l'égalité structurelle (Cf. section 10.4), notée $=_{struct}$.

Algèbre de termes: Je considère ici une algèbre de termes $\mathcal{A}lg_T$ dont constructeur $f \in \mathcal{C}ons_T$ peut être de trois sorte différentes: un constructeur $c \in \mathcal{C}ons$, une primitive $\pi \in \mathcal{P}rim$, ou une variable $x \in \mathcal{V}ars$.

Remarque: notez bien que ici, les variables de la sémantique équationnelle sont considérées comme des constructeurs d'arité nulle pour l'algèbre de termes.

Normalement, la solution classique pour représenter les classes d'équivalence de la relation ($=_{struct}$) est d'utiliser des dags. Ce mot est une abréviation anglaise de *Direct Acyclic Graphs*, ou graphes orientés acycliques. Au lieu de représenter un terme par un arbre, on le représente par un graphe en fusionnant les branches de l'arbre qui sont structurellement égales.

Le problème est qu'ici, la relation R introduit d'autres égalités que ($=_{struct}$). Par exemple, nous aurons à considérer des égalités comme:

$$\begin{array}{l} \alpha.2.p = (cons\ \alpha.1\ \alpha.p) \\ (+\ 1\ 2) = 3 \\ (+\ 1\ 0) = 1 \end{array}$$

Par rapport aux dags, on doit donc considérer des graphes qui peuvent devenir cycliques et qui peuvent porter plusieurs constructeurs d'arités différentes en chaque nœud.

7.3.1 Classes

Chaque classe d'équivalence pour R est numérotée par un entier (il va de soit que c'est un ensemble dénombrable), et je noterai C_k la classe d'équivalence numéro k. Un compteur permet de générer des nouveaux numéros de classe via la fonction *newclass()*.

J'appelle pseudo-terme un ensemble de termes noté $(f\ k_1 \ldots k_n)$ où f est un constructeur de $\mathcal{C}ons_T$, et $k_1 \ldots k_n$ sont des numéros de classe. Le pseudo-terme $(f\ k_1 \ldots k_n)$ représente l'ensemble de termes défini comme suit:

$$(f\ k_1 \ldots k_n) = \{(f\ t_1 \ldots t_n) \mid (\forall i)\ t_i \in C_{k_i}\}$$

Une classe d'équivalence de R sera représentée par un ensemble de pseudo-termes. L'ensemble des termes d'une classe est alors la réunion de ses pseudo-termes.

On utilise une table d'association `Class` pour représenter les classes. Elle associe un numéro de classe à la liste des pseudo-termes qui la constitue. Mais il est utile de considérer aussi une seconde table nommée `Index` et qui constitue l'index de la première et permet de retrouver rapidement (grâce aux techniques de hachage utilisées pour les tables d'associations) à quel numéro de classe appartient un pseudo-terme donné.

La suite de la section est une liste d'algorithmes permettant de gérer ces tables. Étonnement, une astuce bien commode et très efficace pour la complexité des algorithmes est de complètement oublier la table `Class` tant que l'on en a pas explicitement besoin, c'est à dire quand on doit inspecter les pseudo-termes contenus dans une classe. Je me suis aperçu que, en effet, seule la table `Index` est vraiment utile. D'ailleurs, reconstruire `Class` en fonction de `Index` est vraiment facile, et linéaire en le nombre de pseudo-termes:

make_Class ()	met à jour `Class`.
1.	effacer `Class`.
2.	pour chaque entrée (p, k) de la table `Index`, ajouter p dans `Class`$[k]$.

Dans cet algorithme, l'ajout d'un pseudo-terme dans une classe peut se faire en temps constant car les classes d'équivalence garantissent que l'on a pas deux occurrences d'un même pseudo-terme dans toutes les classes.

7.3.2 Insertion d'un terme

Pour insérer un terme de $\mathcal{A}lg_T$ dans les tables, deux algorithmes doivent être définis. Le premier gère l'insertion d'un pseudo-terme, et le deuxième celle d'un terme. Les deux algorithmes renvoient le numéro de la classe correspondante, et modifient la table `Index` en conséquence.

index p	renvoie un numéro de classe pour le pseudo-terme p.
1.	rechercher k = `Index`$[p]$. Si $k \neq \perp$ alors renvoyer k.
2.	sinon il faut une nouvelle classe, donc soit $k' = newclass()$, on pose `Index`$[p]{:=}k'$ et on renvoie finalement k'.
insert t	renvoie un numéro de classe pour le terme $t \in \mathcal{A}lg_T$.
1.	on a $t = (f\ t_1 \ldots t_n)$ avec $f \in \mathcal{C}ons_T$ et $(\forall_i)\quad t_i \in \mathcal{A}lg_T$. Calculer alors récursivement $(\forall i)\quad k_i = insert\ t_i$.
2.	renvoyer $index\ (f\ k_1 \ldots k_n)$.

Cet algorithme d'insertion réalise automatiquement la gestion des classes d'équivalence pour l'égalité structurelle ($=_{struct}$). Insérer un terme est ainsi une opération quasi linéaire en la taille du terme, car le hachage d'un pseudo-terme se fait en temps quasi constant.

Prendre en compte une relation R plus grande nécessite de pouvoir fusionner deux classes.

7.3.3 Fusion de deux classes

J'expose ici un algorithme permettant de fusionner deux classes pour que les termes qu'elles représentent se retrouvent dans une seule et même classe. C'est une opération délicate à plusieurs points de vue. Tout d'abord, Il faut se méfier de la ribambelle des fusions qui peuvent être induites en cascade par une fusion initiale. En effet, pour fusionner les classes k_1 et k_2, je remplace toute occurrence de l'une d'elle (je choisirai k_2) par l'autre (donc k_1). Or, si on a par exemple quatre classes $C_1 \ldots C_4$ telles que $C_1 = \{\ (f\ 3)\ \}$ et $C_2 = \{\ (f\ 4)\ \}$ et que l'on fusionne maintenant C_3 et C_4, alors les classes C_1 et C_2 doivent être fusionnées également car elles contiennent toutes les deux un terme identique.

Maintenant, le problème est de faire cette fusion en un temps raisonnable. Pour y parvenir, je suggère d'utiliser une table d'association qui conserve en permanence une trace des fusions effectuées. Cela permet de factoriser la tâche longue et fastidieuse qui consiste à remplacer toute les occurrences d'un numéro de classe périmé (k_2) par son nouveau numéro (k_1).

J'appelerai donc `Equal` la table qui associe à tout numéro de classe périmé sont nouveau numéro. On peut alors définir aisément une fonction *normalize* qui met à jour tous les numéros de classe périmés contenus dans son argument[1]. Enfin pour gérer la cascade des fusions, il sera commode d'utiliser une file d'attente `F` des fusions à effectuer.

1. Pour simplifier, je masque ici le problème de la surcharge de cette fonction *normalize*. En pratique, il en faut une pour chaque type d'argument, mais c'est un détail d'implémentation qui peut dépendre du langage choisi.

fusion k_1 k_2	réalise la fusion de k_1 et k_2.
1.	initialiser F avec F:=[(k_1, k_2)]
2.	tant que $F \neq []$, retirer une fusion (k_1, k_2) de F et exécuter l'algorithme (*make* k_1 k_2).

make k_1^0 k_2^0	fait une étape de fusion et génère les fusions en cascade.
1.	soient $k_i = normalize\ k_i^0$ pour $i = 1, 2$.
2.	si $k_1 = k_2$ ne rien faire d'autre.
3.	effectuer `Equal`$[k_2]$:=k_1 et pour les autres entrées effectuer `Equal`$[k]$:=*normalize* (`Equal`[k]).
4.	créer une table d'association `Tmp` initialement vide et du même type que `Index`. Elle sera destinée à remplacer cette dernière.
5.	pour chaque entrée (p, k) dans la table `Index`:
5.1.	calculer $(p', k') = normalize\ (p, k)$ et rechercher k_{tmp} = `Tmp`$[p']$.
5.2.	si $k_{tmp} = \perp$, alors `Tmp`$[p']$:=k'.
5.3.	sinon ajouter la fusion (k', k_{tmp}) dans F.
6.	remplacer `Index` par `Tmp`.

La complexité finale de la fusion de deux classes peut être parfois élevée. Si aucune fusion en cascade ne se produit, elle est linéaire en le nombre de pseudo-terme, donc plutôt faible. Il faut en pratique la multiplier par le nombre de fusion en cascade produites jusqu'à ce que F soit vide. Or, le nombre de fusion en cascade peut, en théorie, atteindre le nombre de classes présentes au départ. Mais ce cas extrême correspond à l'effondrement de tous les termes en une seule classe d'équivalence, ce qui n'arrive pour ainsi dire jamais. En pratique, la fusion reste donc quasi linéaire en le nombre de pseudo-termes, car on ne verra qu'un nombre assez faible de fusions en cascade (deux ou trois en moyenne!).

7.3.4 Fusion de deux termes

Si deux termes sont en relation par R, il faut que leur classes d'équivalence n'en forment qu'une seule. On doit donc les fusionner. On commence donc par insérer l'un et l'autre terme, et on fusionne leur numéros de classes. Cependant, je veux signaler ici une optimisation qui en pratique s'avère bien utile.

Dans certains cas, on introduit dans l'ensemble des classes un nouveau pseudo-terme juste avant de le fusionner avec une autre classe. On peut alors alléger la complexité de l'insertion en évitant de construire la classe correspondant au nouveau pseudo-terme. On évite ainsi de devoir utiliser l'algorithme de fusion, car il suffit d'ajouter le pseudo-terme à la classe avec laquelle il doit fusionner.

Par exemple, supposons que l'on doive fusionner (x) avec $(+\ 1\ 2)$ dans un contexte où $\texttt{Index}[x] = \bot$ et $\texttt{Index}[(+\ 1\ 2)] = C_4$. Alors il n'est pas nécessaire de créer une classe pour x pour ensuite la fusionner avec C_4: il suffit d'ajouter (x) dans la classe C_4 et de mettre à jour la table `Index`.

J'utilise cette technique lors de la fusion de deux termes, mais je ne donne pas ici l'algorithme d'insertion, car il est assez fastidieux. Il faut en effet prendre en compte de nombreux cas, mais il ne recèle aucune difficulté.

7.3.5 Modifications pour les mécanismes de démons

Afin de préparer les calculs de la section 7.5, je signale que les fonction *insert* et *make* subissent en pratique les modifications suivantes.

- une variable globale X est initialisée à l'ensemble vide.
- chaque fois que la fonction *insert* est appelée avec une variable x qui n'est pas dans X, alors on ajoute x à X, et on appelle la fonction *newvar* x qui sera définie plus section 7.5.
- chaque fois que l'on ajoute un nouveau pseudo terme dans une classe k, on appelle la fonction *modify* k qui sera définie section 7.5.

7.4 Démons

Dans cette section, je vais présenter les différents démons associés à la relation $\rightarrow_{\mathcal{P}rg}$. Par soucis de cohérence, et dans la mesure où cette section est essentiellement technique, je présenterai aussi dans cette section les démons associés aux extensions de $\rightarrow_{\mathcal{P}rg}$ nécessaires aux transformations de programmes et vues au chapitre 5. Je signale enfin qu'aucun démon ne correspond aux substitutions étant donné qu'elles sont automatiquement gérées par les classes représentant les systèmes d'équations (voir section 7.3).

Chaque section qui suit traite d'un type de démon. Pour chacun, je donne la forme de la variable qu'il produit et son exécution. Je fais librement référence aux notions de classes et de pseudo-termes définis au début de la section 7.3. Enfin, les démons peuvent se mettre en attente sur une ou plusieurs classes, via la fonction *sleep* qui est décrite à la section 7.5.

7.4.1 Sous-terme

Variable:	$x.j$, avec $j \geq 0$.
Démon:	
1.	soit k le numéro de la classe d'équivalence de x, et k' celui de $x.j$.
2.	pour tout pseudo-terme $(f\ k_1 \ldots k_n)$ dans C_k tel que $f \in \mathcal{C}ons$ (et non pas $\mathcal{C}ons_T$), faire fusionner les classes $C_{k'}$ et C_{k_j}.
3.	si aucune fusion n'est trouvée, mettre le démon en attente sur k.

7.4.2 Programme

Dans cette section, je présente en fait quatre démons qui sont construits sur le même schéma. Je commence donc par la description d'un démon générique appelé *prog*.

prog x y	fonction qui applique une équation de la forme $y = t$ sur la variable x.
1.	soit k le numéro de la classe de x.
2.	pour tout pseudo-terme $(c\ \ldots) \in C_k$ et tout terme t tel que $c \rightarrow y = t$ soit une équation du programme $\mathcal{P}rg$, alors fusionner les termes $y[x]$ et $t[x]$.
3.	si aucun pseudo-terme de la forme $(c\ \ldots)$ n'est trouvé dans C_k, mettre le démon en attente sur k.

Les quatre démons sont alors:

Variable:	$x.r$, avec $r \in \mathcal{R}es$
Démon:	*prog* x $\alpha.r$
Variable:	$x.n.p$, avec $p \in \mathcal{P}rm$, $n \geq 0$
Démon:	*prog* x $\alpha.n.p$
Variable:	$x.L_n.p$, avec $p \in \mathcal{P}rm$, $n \geq 0$
Démon:	*prog* x $\alpha.L_n.p$
Variable:	$x.L_n$, avec $n \geq 0$
Démon:	*prog* x $\alpha.L_n$

7.4.3 Spécialisation

Pour la transformation, se reporter à la section 5.3. J'utilise ici les tables définies à la section 5.9.1 qui permettent de gérer efficacement des spécialisations multiples.

Pour chaque entrée dans la table (C, S), on peut définir des démons. Je donne ici l'algorithme qui, pour le contexte C et un couple $(r, r') \in S$ permet de spécialiser une variable de la forme $x.r$ en $x.r'$ si le contexte C est vérifié:

Variable:	$x.r$ avec $(C, S) \in SPE$ et $(r, r') \in S$
Démon :	
1.	tester pour chaque couple $(p, v) \in C$ si $x.p$ et v sont dans la même classe.
2.	le cas échéant, fusionner les termes $x.r$ et $x.r'$. Sinon, mettre le démon en attente sur au moins une des classes ayant posée problème.

7.4.4 Composition

Pour la transformation, voir section 5.4. J'utilise les tables données dans la section 5.9.2. Je définis alors, pour toute entrée de la forme $r_1 \circ r_2 = r_3$ dans la table de composition, le démon suivant:

Variable:	$x.r_2$
Démon:	
1.	Soit k la classe de x.
2.	Si il existe une variable $y.r_1$ dans la classe k:
2.1	créer la nouvelle variable locale $\alpha.L_M$.
2.2	insérer dans les classes les équations suivantes:
	$$\left\{\begin{array}{ccll} \alpha.L_M & = & y & \\ \alpha.L_M.p & = & y.p & (\forall p \in \mathcal{D}ep(r_1)) \\ y.r_1 & = & \alpha.L_M.r_1 & \\ \alpha.L_M.q' & = & x.q & \text{avec } r_1 \circ q = q' \\ x.r_2 & = & \alpha.L_M.r_3 & \end{array}\right\}$$
3.	Sinon, mettre le démon en attente sur la classe k.

7.4.5 Précalculation

Pour la transformation, voir section 5.5. J'utilise les tables données dans les sections 5.9.2 et 5.9.3. Je définis alors, pour toute entrée de la forme $p \circ r = r'$ dans la table de composition, le démon suivant:

Variable:	$x.r$
Démon:	
1.	Soit k la classe de x et $(c \ldots)$ un pseudo-terme de la classe de α.
2.	Si $(p, r) \in PreCalc[c]$ terminer l'exécution du démon.
3.	Si k est aussi la classe de $\alpha.p$, alors:
3.1	créer la nouvelle variable locale $\alpha.L_M$.
3.2	insérer dans les classes les équations suivantes:
	$$\left\{\begin{array}{ccll} x.r & = & \alpha.r' & \text{avec } p \circ r = r' \\ \alpha.q' & = & x.q & (\forall q \in \mathcal{D}ep(r) \\ & & \text{et } p \circ q = q' & \end{array}\right\}$$
3.	Sinon, mettre le démon en attente sur la classe k.

7.4.6 Élimination des identités

Pour la transformation, voir la section 5.6. Étant donné l'ensemble I_∞ qui y est défini, je définis le démon suivant:

Variable:	$x.r$ avec $(p, r) \in I_\infty$
Démon:	pour tout p tel que $(p, r) \in I_\infty$, fusionner $x.p$ et $x.r$.

7.5 Calcul du Point Fixe

Je reprends maintenant la solution générale exposée à la section 7.2. Je rappelle brièvement que le but est de calculer les suites Σ_n, X_n avec un mécanisme de démons, les systèmes étant représentés par des classes de pseudo-termes.

L'algorithme que je propose est assez complexe dans le sens où une partie de ces fonctionnements sont cachés dans les démons. Le principe est d'itérer une fonction, *Step*, jusqu'à obtention d'un point fixe. Le problème de savoir quand ce point fixe est atteint sera traité dans la section suivante.

Je note DR l'ensemble des démons réveillés et DA la table qui donne pour toute classe les démons en attente sur cette classe.

Comme annoncé dans la section 7.3, à chaque fois que la classe k risque d'être modifiée, la fonction *modify* k est appelée. Je peux maintenant donner le corps de cette fonction:

modify k	signale que la classe k est modifiée.
1.	faire $D{:=}D \cup DA[k]$
2.	initialiser $DA[k]{:=}\emptyset$

De même, comme prévu à la section 7.3, chaque fois qu'une nouvelle variable x est insérée dans les classes, la fonction suivante est *newvar* x est appelée, J'en donne maintenant la description:

newvar x	signale que x est une nouvelle variable.
1.	pour chaque démon d définissant la variable x, faire $D{:=}D \cup \{d\}$.
2.	si x est de la forme $y.r$ avec $r \in \mathcal{R}es$, alors pour toute variable $z \in \{y\} \cup \{y.p \mid p \in \mathcal{D}ep(r)\}$, faire *insert* z pour insérer la variable z dans les classes d'équivalence.

Pour terminer, la fonction *sleep* d K qui met en attente le démon d sur les classes K est simplement définie par:

sleep d K	pour tout $k \in K$, faire $DA[k]{:=}DA[k] \cup \{d\}$

Enfin, je peux maintenant donner le corps de la fonction *Step*:

Step ()	calcule une itération du point fixe.
•	Si DR n'est pas vide:
1.	choisir $d \in DR$ et faire $DR{:=}DR - \{d\}$.
2.	exécuter le démon d.
3.	renvoyer `true`.
•	Sinon renvoyer `false`.

Le calcul du point fixe

L'idée de base consiste bien sûr à itérer la fonction *Step* tant qu'elle renvoie `true`. Mais en pratique, cette solution n'est pas raisonnable pour un problème de non-terminaison. De plus, les démons de se préoccupent pas des relation de réécriture sur les primitives, qu'il faut pourtant bien prendre en compte. Je vais donc examiner ces deux points avant de fournir l'algorithme de calcul d'un “point fixe”.

Terminaison: je vais me permettre ici de donner mon opinon sur les problèmes de terminaison dans le cadre des transformations de programme. Tout d'abord, une simple remarque. Imaginons un programme dont l'évaluation ne termine pas. Autrement dit, Σ_∞ n'est jamais atteint. Alors l'évaluation partielle ne pourra jamais terminer, quelque soit le langage, le formalisme et la méthode, à moins de ne considérer que des programmes qui terminent. Or la terminaison étant indécidable, on *veut* traiter des programmes qui peuvent ne jamais s'arrêter.

Par ailleurs, même en imaginant qu'un algorithme termine, qu'il soit polynomial ou exponentiel, qu'importe: il peut être de toute manière trop long à calculer!

Donc, en ce qui me concerne, la question est claire: on ne peut pas itérer indéfiniment la fonction *Step*. En revanche, elle finit toujours par renvoyer `false` si le programme termine, et de toute manière, il est utile de limiter arbitrairement le nombre d'itérations de cette fonction pour des raisons pratiques.

Réécriture: L'idée de base est d'alterner la fonction *Step* avec une fonction qui gère les règles de réécriture (fonction *reec*, Cf. section 7.6). Malheureusement, malgré mes efforts appeler cette fonction reste une opération assez coûteuse en temps par rapport au reste. Il faut donc “délayer” l'utilisation des règles de réécritures et les effectuer le moins souvent possible.

Version finale: la fonction $(\mathit{Fix}\ m\ X\ \Sigma)$ calcule un système Σ' tel que $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$, en appliquant *Step* au plus m fois. A la fin, Σ' contient essentiellement des équations relatives aux variables X.

$(Fix\ m\ X\ \Sigma)$	
1.	initialiser les classes.
2.	initialiser les démons: $DR = \emptyset$, $DA = \emptyset$.
3.	pour toute équation $x = t \in \Sigma$, faire *fusion* $x\ t$.
4.	pour toute variable $x \in X$, faire *insert* x.
4.	tant que $m > 0$:
4.1	Si *Step* () renvoie `false`, appeler *reec*.
4.2	Si $DR = \emptyset$ alors m:=0 sinon m:=$m - 1$.
5	renvoyer *out* ().

7.6 Primitives & Unification

Dans cette section, je vais présenter la fonction *reec* qui permet d'appliquer les règles de réécriture sur les primitives ($\triangleright$) sur un ensemble de classe d'équivalence.

Je considère ici deux types de règles de réécriture, celles qui traitent des opérations primitives par exemple sur les entiers, et celles qui sont données sous forme de patterns (Cf. section 10.4).

Pour gérer les opérations élémentaires sur les entiers (par exemples) il suffit de rechercher, dans toute classe k, les pseudo-termes de la forme $(\pi\ k_a\ k_b)$ oú $\pi \in \{+, -, \ldots\}$. Ensuite, pour chaque entier a présent dans la classe k_a et pour chaque entier k_b présent dans la classe k_b, on calcule en fonction de π de a et b le résultat de l'opération qui est un entier c. Il suffit alors de fusionner les classes k et *index* c.

Pour gérer les règles de réécriture qui sont présentées sous forme de pattern, un mécanisme d'unification adapté doit être défini. Prenons un pattern p et ses variables $a_1 \ldots a_n$. Pour unifier p avec une classe k, je vais travailler avec des ensembles de substitutions.

Soit S_1 et S_2 deux ensembles de substitutions, je définis leur fusion par:

$$S_1 \vee S_2 = \{\sigma_1 \sqcup \sigma_2 \mid (\forall i)\sigma_i \in S_i\}$$

avec la même définition pour $\sqcup$ que dans la section 10.4. On peut d'ailleurs remarquer que l'algorithme de calcul des bases d'ensembles coniques est applicable (en inversant l'ordre). On a alors $S_1 \vee S_2 = base\ (S_1 \cup S_2)$.

Soit alors la fonction *occurences* $p\ k$ qui cherche l'ensemble des substitutions σ telles que $\sigma(p) \in C_k$, où C_k est la classe numéro k. On peut la définir et la calculer ainsi:

occurences p k	cherche des substitutions pour le pattern p et la classe k.
•	Si p est une variable a, renvoyer $\{[a{:=}k]\}$.
•	Sinon, $p = (\pi\ p_1 \dots p_n)$. Renvoyer: $$\bigvee \{occurences\ p_i\ k_i \mid i \leq n\}$$

Dès lors, il est facile de définir la fonction *reec*. Je ne détaille pas la partie relative aux opérations élémentaires, mais il suffit de la rajouter:

reec ()	effectue les fusions dues à ($\triangleright$).
1.	effectuer les opérations élémentaires.
2.	pour tout $p_1 \triangleright p_2$, et pour tout k:
2.1	calculer $S = occurences\ p_1\ k$
2.2	pour tout $\sigma \in S$, faire *fusion* k $(index\ \sigma(p_2))$.

C'est une opération relativement coûteuse, surtout si il existe des patterns de grande profondeur. Mais en pratique, c'est assez rapide.

7.7 Système Final

Ici, le problème que j'aborde concerne la restitution d'un système d'équation à partir des ensembles de classes. Le principal problème est bien entendu de rendre un système qui soit le plus concis possible. En effet, il y a dans la structure des classes beaucoup de sous-termes qui sont partagés par plusieurs équations.

Le but va être ici de rendre compte et de profiter de ce partage (cette factorisation) par des nouvelles variables locales. L'idée de base pour cette transformation est bien entendu d'attribuer à chaque classe d'équivalence C_k une nouvelle variable locale $\alpha.L_{m_k}$ que je note x_k pour simplifier.

Soit donc un pseudo-terme dans une classe C_k. Si c'est une variable x, on peut produire une équation de la forme $x = x_k$ ou bien de la forme $x_k = x$. En fonction de la forme précise de x, on pourra choisir la bonne alternative.

Soit maintenant un pseudo-terme $(f\ k_1 \dots k_n)$ dans la classe C_k, on a pas le choix, la seule équation possible est $x_k = (f\ x_{k_1} \dots x_{k_n})$.

Pour regrouper ces deux résultats, je définis donc deux fonctions, *left* et *right* qui prennent une classe en paramètre et renvoient respectivement un ensemble de variables et de termes qui correspondent aux parties gauches et droite des équations:

$$\begin{array}{lcl}
\mathit{left}(k) & = & \{\alpha.r \in C_k \mid r \in \mathcal{R}es\} \\
 & \cup & \{\alpha.L_m.p \in C_k \mid p \in \mathcal{P}rm\} \\
\mathit{right}(k) & = & \{\alpha.p \in C_k \mid r \in \mathcal{P}rm\} \\
 & \cup & \{\alpha.L_m.r \in C_k \mid r \in \mathcal{P}rm\} \\
 & \cup & \mathcal{X}_{loc} \cap C_k \\
 & \cup & \{(f\ x_{k_1} \dots x_{k_n}) \mid (f\ x_1 \dots x_n) \in C_k\}
\end{array}$$

Le système final associé à un ensemble de classe $C_0 \dots C_m$ est alors construit comme suit:

$$\begin{array}{lcl}
\Sigma_\infty = & = & \{x = x_k \mid x \in \mathit{left}(k) \quad k \leq m\} \\
 & \cup & \{x_k = t \mid t \in \mathit{right}(k) \quad k \leq m\}
\end{array}$$

On peut bien entendu faire de nombreuses optimisations pour éviter certaines redondances. Par exemple, quand *left* est réduit à un seul élément, on peut se passer de la variable locale x_k. Quand il n'y a qu'un terme dans *right* également. Je ne détaille pas ici ces optimisations fastidieuses mais importantes.

Une autre optimisation, plus difficile mais également plus intéressante, consiste à regrouper autant que faire ce peut les calculs sur les sous-termes. Imaginons que nous ayons les classes suivantes:

$$\begin{array}{l}
C_0 = \{\alpha.L_1\ \alpha.L_2\ \alpha.1\} \\
C_1 = \{\alpha.L_1.p\ \alpha.p\} \\
C_2 = \{\alpha.L_1.r\ \alpha.r\} \\
C_3 = \{\alpha.L_2.p'\ \alpha.p'\} \\
C_4 = \{\alpha.L_2.r'\ \alpha.r'\}
\end{array}$$

Normalement, voici le système Σ_∞ que l'on doit obtenir, avec les optimisations précédentes:

$$\left\{\begin{array}{ccl}
\alpha.r & = & \alpha.L_1.r \\
\alpha.r' & = & \alpha.L_2.r' \\
\alpha.L_1.p & = & \alpha.p \\
\alpha.L_2.p' & = & \alpha.p' \\
\alpha.L_1 & = & \alpha.1 \\
\alpha.L_2 & = & \alpha.1
\end{array}\right\}$$

Maintenant, je suppose de plus que dans le programme $\mathcal{P}rg$, il n'y a pas encore d'équation de la forme $c \to \alpha.1.p = t$ ou $c \to \alpha.1.p' = t$ (avec t quelconque). Alors comme $p \neq p'$ et $r \neq r'$, on a pas besoin des variables locales $\alpha.L_1$ et $\alpha.L_2$. Donc on peut produire directement le système suivant:

$$\left\{\begin{array}{ccl}
\alpha.r & = & \alpha.1.r \\
\alpha.r' & = & \alpha.1.r' \\
\alpha.1.p & = & \alpha.p \\
\alpha.1.p' & = & \alpha.p'
\end{array}\right\}$$

Ce type d'optimisation n'est pas très difficile à implémenter à l'aide la relation de dépendance $\mathcal{D}ep$. Il suffit de récolter toutes les dépendances entre $x.p$

et $x.r$ et de les classer suivant la classe d'équivalence de x. Celles que l'on peut regrouper sont alors assignée à une nouvelle variable globale unique, ou alors, à une variable $\alpha.k$ si possible.

Chapitre 8

Sémantique Opérationnelle

Ce chapitre est consacré au calcul de la sémantique opérationnelle d'un programme équationnel. C'est une opération capitale pour plusieurs raisons. Tout d'abord, elle permet d'utiliser les programmes équationnels comme une représentation effective d'un algorithme. J'entends ici par effective la possibilité de transformer un programme équationnel en une forme plus classique et exécutable.

Dans le cas général, il n'existe pas de stratégie qui puisse recouvrir la relation $[\![\mathcal{P}rg]\!]$, même pour un champs d'observation défini à l'avance. Je dois donc me limiter à une sous-relation de $[\![\mathcal{P}rg]\!]$, et je consacre la section 8.2 à expliquer en quoi cette restriction est suffisante et en quoi elle est légitime. Le lecteur qui ne ressent pas le besoin de cette justification peut poursuivre sa lecture directement aux sections suivantes.

Avant d'aborder l'algorithme de calcul des stratégies dans son ensemble, je commence par quelques travaux préparatifs (section 8.3.1), essentiellement relatifs aux séquences de pas élémentaires. Ensuite, j'expliquerai comment je calcule un champs d'observation maximal (section 8.3.2) qui corresponde à une sémantique opérationnelle du programme considéré.

8.1 Exemple

Dans la section 3.5, je m'étais empressé d'oublier la structure des dérivations $\rightarrow_{\mathcal{P}rg}$ qui composent la relation $\Rightarrow_{\mathcal{P}rg}$. Si c'est un point de vue intéressant pour introduire une nouvelle forme de transformation de programme, il faut parfois avoir une façon canonique de résoudre les systèmes d'équations. Par exemple, c'est le cas quand on veut revenir à un programme fonctionnel.

Cette section est une introduction à la notion de *stratégie*, qui n'est qu'une forme particulière de dérivations $\Rightarrow_{\mathcal{P}rg}$ pour lesquelles l'ordre d'application des équations de programme est figé.

Pour cette introduction, je m'appuierai encore une fois sur l'exemple de l'inversion d'une liste. Je met en garde le lecteur, car cette introduction ne concerne que la notion de stratégie, et je ne donnerai aucune indication sur la technique permettant de calculer ces stratégies. C'est en effet une opération assez complexe nécessitant ses propres approches intuitives et une bonne dose d'algorithmique. Je consacrerai le chapitre 8 aux algorithmes qui permettent de calculer un ensemble de stratégies pour un programme équationnel. J'ajoute juste pour finir que c'est grâce à la puissance de ces derniers algorithmes que l'ensemble de ma méthode de transformation de programmes peut fonctionner en pratique.

A propos de "définir": quelques formules me seront bien utiles pour simplifier la présentation des stratégies. Ainsi, je dirai qu'un système *définit* une variable x quand il contient une équation de la forme $x = v$ où v est une valeur. Cela signifie qu'il peut contenir de nombreuses autres équations, notamment de la forme $x = t$ où t n'est pas une valeur.

8.1.1 Un exemple de Stratégie

Je reprends donc l'exemple de l'inversion d'une liste avec des notations un peu différentes pour simplifier et en notant π_1, π_2 et π_3 les propriétés du programme pour pouvoir y faire référence plus tard:

$$\left\{ \begin{array}{ccll} \mathit{cons} & \rightarrow & \alpha.r = \alpha.2.r & (\pi_1) \\ \mathit{cons} & \rightarrow & \alpha.2.p = (\mathit{cons}\ \alpha.1\ \alpha.p) & (\pi_2) \\ \mathit{nil} & \rightarrow & \alpha.r = \alpha.p & (\pi_3) \end{array} \right\}$$

Je me permet de rappeler la sémantique de ce programme. Pour toute liste l, si l' est la liste l inversée, alors on a:

$$(l, \mathit{nil})[\![\mathcal{P}rg]\!]_{p,r}(l')$$

Ceci signifie par définition qu'il existe une dérivation du système initial Σ_p au système final Σ_r:

$$\Sigma_p = \left\{ \begin{array}{ccc} \alpha & = & l \\ \alpha.p & = & (\mathit{nil}) \end{array} \right\} \quad \Rightarrow_{\mathcal{P}rg} \quad \Sigma_r = \left\{ \begin{array}{ccc} \alpha.r & = & l' \end{array} \right\}$$

Je m'intéresse ici à donner une structure canonique à la dérivation $\Sigma_p \Rightarrow_{\mathcal{P}rg} \Sigma_r$. Je vais donc noter $\stackrel{p,r,x}{\rightarrow}$ une relation qui définit $x.r$ à partir de $x.p$. Autrement dit $\Sigma_1 \stackrel{p,r,x}{\rightarrow} \Sigma_2$ signifie que Σ_1 définit les variables x et $x.p$, que Σ_2 définit la variable $x.r$ sur x et bien entendu que $\Sigma_1 \Rightarrow_{\mathcal{P}rg} \Sigma_2$. La structure qui m'intéresse se limite à l'ordre dans lequel les équations du programme sont instanciées.

Dans le cas particulier où une propriété $\pi \in \mathcal{P}rg$ est instanciée sur la variable x, je noterai la relation $\rightarrow_{\mathcal{P}rg}$ par $\stackrel{\pi,x}{\rightarrow}$. En revanche, une dérivation qui n'instancie jamais de propriété du programme sera tout simplement notée $\Rightarrow$. Je cherche donc à écrire la relation $\stackrel{p,r,x}{\rightarrow}$ comme une succession de $\stackrel{\pi,x}{\rightarrow}$ et de $\Rightarrow$.

On peut remarquer que la seule occasion où $\rightarrow_{\mathcal{P}rg}$ fait apparaître une variable de la forme $x.r$ est celle où la propriété π_1 ou π_3 est instanciée sur la variable

x. Ce qui suggère immédiatement que si $\Sigma_1 \overset{p,r,x}{\rightarrow} \Sigma_2$, alors on a nécessairement $x = (cons\ v_1\ v_2)$ ou bien $x = (nil)$. Maintenant:

- Si $x = (nil)$, alors seule l'équation de programme π_3 peut s'appliquer. Or comme Σ_1 définit la variable $x.p$, une simple substitution permet de définir la variable $x.r$. Je peut donc définir la relation $\overset{p,r,x}{\rightarrow}$, dans le cas $x = (nil)$, par:
$$\overset{p,r,x}{\rightarrow} = [\overset{\pi_3}{\rightarrow}; \Rightarrow]$$
- Si $x = (cons\ v_1\ v_2)$, alors c'est un peu plus compliqué. Considérons les deux équations que l'on peut appliquer. La première, π_1, permet de définir $x.r$ en fonction de $x.2.r$ et la seconde, π_2, permet de définir $x.2.p$ en fonction des variables $x.p$ et $x.1$. Or $x.1$ tout comme $x.2$ peuvent être définies par une dérivation sous-terme et $x.p$ est supposée définie. Donc, pour définir $x.r$ en fonction de x et de $x.p$, il ne reste plus qu'un "trou" entre $x.2.p$ et $x.2.r$. Pour combler ce trou, j'utilise récursivement une dérivation de la forme $\overset{p,r,x.2}{\rightarrow}$ et le tour est joué. Je peut donc définir la relation $\overset{p,r,x}{\rightarrow}$ dans le cas où $x = (cons\ v_1\ v_2)$ par:
$$\overset{p,r,x}{\rightarrow} = [\overset{\pi_2}{\rightarrow}; \Rightarrow; \overset{p,r,x.2}{\rightarrow}; \overset{\pi_1}{\rightarrow}; \Rightarrow]$$

Pour définir complètement la relation $\overset{p,r,x}{\rightarrow}$, il suffirait alors d'écrire:

$$\overset{p,r,x}{\rightarrow} = [\overset{\pi_2}{\rightarrow}; \Rightarrow; \overset{p,r,x.2}{\rightarrow}; \overset{\pi_1}{\rightarrow}; \Rightarrow] + [\overset{\pi_3}{\rightarrow}; \Rightarrow]$$

Une façon d'écrire cette dérivation de manière plus concise est:

$$[p, r, \alpha] = [\pi_2; p, r, \alpha.2; \pi_1] + [\pi_3]$$

Mais sous cette forme, on perd l'information qui associe la première partie de la somme au constructeur *cons* et la deuxième au constructeur *nil*. De plus, le fait que ces deux séquences sont associées à la paire (p, r) est un noyé dans la masse. C'est pourquoi je note cette relation sous la forme d'un ensemble de séquences, que j'appelle *stratégies* et que je note $\mathcal{S}$ qui renferme toutes ces informations:

$$\mathcal{S} = \{ \quad (p, r, cons, seq_1) \quad ; \quad (p, r, nil, seq_2) \quad \}$$

avec $seq_1 = [\pi_2; p, r, \alpha.2; \pi_1]$ et $seq_2 = [\pi_3]$. Cet ensemble se lit ainsi: on peut calculer $x.r$ en fonction de $x.p$ si $x = (cons\ v_1\ v_2)$ avec la séquence seq_1 et de même si $x = (nil)$ mais cette fois ci avec la séquence seq_2. Une telle stratégie sera aussi appelé *sémantique opérationnelle* du programme $\mathcal{P}rg$ et notée $[\![\mathcal{P}rg]\!]^{op}$.

8.1.2 Sémantique Opérationnelle (définition)

Pas élémentaire: c'est ou bien une équation $y = t$ du programme $\mathcal{P}rg$, ou bien un triplet (P, R, x) avec $P \subset \mathcal{P}rm$, $R \subset \mathcal{R}es$ et $x \in \{\alpha.k,\ \alpha.L_k\}$.

Séquence: c'est une liste ordonnée de pas élémentaires $seq = [s_1; \ldots; s_n]$.

Stratégie: c'est un ensemble de quadruplets (P, R, c, seq), avec $P \subset \mathcal{P}rm$, $R \subset \mathcal{R}es$, $c \in \mathcal{C}ons$ et seq une séquence.

En conjonction avec ces définitions, je définis – récursivement – les relations suivantes.

- Une dérivation calculant R à partir de P sur x avec la stratégie $\mathcal{S}$ est définie par:

$$\begin{array}{rcl} \Sigma \overset{P,R,x}{\to_{\mathcal{S}}} \Sigma' & \Leftrightarrow & x = (c \dots) \in \Sigma \\ & \text{et} & (P', R, c, seq) \in \mathcal{S} \quad P' \subset P \\ & \text{et} & \Sigma \overset{seq,x}{\to_{\mathcal{S}}} \Sigma' \end{array}$$

- L'application d'une séquence de pas élémentaires seq sur la variable x est définie par:

$$\begin{array}{rcl} \Sigma \overset{seq,x}{\to_{\mathcal{S}}} \Sigma' & \Leftrightarrow & seq = [s_1; \dots; s_n] \\ & \text{et} & \Sigma \overset{s_1,x}{\to_{\mathcal{S}}} \dots \overset{s_n,x}{\to_{\mathcal{S}}} \Sigma' \end{array}$$

- L'instanciation d'une équation $y = t$ sur la variable x est définie par:

$$\Sigma \overset{y=t,x}{\to_{\mathcal{S}}} \Sigma' \quad \Leftrightarrow \quad \Sigma' = \Sigma \cup \{y[x] = t[x]\}$$

- L'appel récursif du pas élémentaire (P, R, y) sur la variable x est défini par:

$$\Sigma \overset{(P,R,y),x}{\to_{\mathcal{S}}} \Sigma' \quad \Leftrightarrow \quad \Sigma \overset{P,R,y[x]}{\to_{\mathcal{S}}} \Sigma'$$

On peut remarquer que les dérivations $\overset{P,R,x}{\to}$ sont incluses dans les dérivations $\Rightarrow_{\mathcal{P}rg}$. Ce sont donc bien des dérivations particulières issues de la sémantique équationnelle. Mais, remarquez aussi que l'ordre d'application des équations du programme $\mathcal{P}rg$ y est rigoureusement contrôlé et ordonné par un schéma récursif.

Quand une stratégie $\mathcal{S}$ permet de remplacer toutes les dérivations du type $\Sigma_P \Rightarrow_{\mathcal{P}rg} \Sigma_R$ avec $(P, R) \in \mathcal{O}bs$, je dirai que l'on a une sémantique opérationnelle de $\mathcal{P}rg$ pour le champs d'observation $\mathcal{O}bs$. Ainsi, je noterai:

$$[\![\mathcal{P}rg]\!]^{op}_{\mathcal{O}bs} = \mathcal{S}$$

et je pourrai définir la relation suivante:

$$\begin{array}{c} (v, v_1 \dots v_n) \quad [\![\mathcal{P}rg]\!]^{op}_{P,R} \quad (w_1 \dots w_m) \\ \Leftrightarrow \\ \left\{ \begin{array}{l} \alpha = v \\ \alpha.p_1 = v_1 \\ \dots \\ \alpha.p_n = v_n \end{array} \right. \quad \overset{P,R,\alpha}{\to_{\mathcal{S}}} \quad \left\{ \begin{array}{l} \alpha.r_1 = w_1 \\ \dots \\ \alpha.r_m = w_m \end{array} \right. \end{array}$$

Le lecteur pourra comparer avec la sémantique équationnelle de $\mathcal{P}rg$ dont je rappelle ici la définition:

$$
\begin{array}{ccc}
(v, v_1 \dots v_n) & [\![\mathcal{P}rg]\!]_{P,R} & (w_1 \dots w_m) \\
\Leftrightarrow & & \\
\left\{ \begin{array}{l} \alpha = v \\ \alpha.p_1 = v_1 \\ \dots \\ \alpha.p_n = v_n \end{array} \right. & \Rightarrow_{\mathcal{P}rg} & \left\{ \begin{array}{l} \alpha.r_1 = w_1 \\ \dots \\ \alpha.r_m = w_m \end{array} \right.
\end{array}
$$

La différence réside uniquement dans le fait que, dans la première sémantique, la relation $\overset{P,R,\alpha}{\to_{\mathcal{S}}}$ fait apparaître la structure récursive de la dérivation, alors que dans la deuxième sémantique, cette structure est cachée par la relation $\Rightarrow_{\mathcal{P}rg}$.

8.2 Incomplétude

Je rappelle tout d'abord l'objectif. Étant donné un programme équationnel $\mathcal{P}rg$, qui est fixé pour tout le chapitre, on cherche une stratégie $\mathcal{S}$ et un champs d'observation $\mathcal{O}bs$, tels que pour $(P, R) \in \mathcal{O}bs$, on ait:

$$[\![\mathcal{P}rg]\!]_{(P,R)} = \overset{P,R,\alpha}{\to_{\mathcal{S}}}$$

Évidemment, il est possible que la relation $[\![\mathcal{P}rg]\!]_{(P,R)}$ soit non vide pour certaines paires (P, R) sans qu'il puisse exister de stratégie adéquate. On risque alors de perdre une grande partie de la sémantique en cours de route.

Une réaction possible à ce problème est de l'oublier complètement: en effet, toutes les paires (P, R) ne sont pas nécessaires. A la fin de mes calculs, j'obtiendrais un champs d'observation pour lesquelles on a l'égalité. Pourvu que l'on trouve la ou les paires qui nous intéressent pour un programme donné, cela sera amplement suffisant. Or on verra au chapitre 9 que les programmes issus d'une traduction fonctionnelle contiennent toujours au moins une stratégie. Les transformations de programmes conservant toujours les programmes initiaux, cette stratégie initiale sera donc toujours présente. En un sens, on ne peut qu'espérer en trouver de meilleures!

Une deuxième réaction consiste à se demander pourquoi il est pertinent de rechercher une sémantique opérationnelle sous forme de stratégie, et pourquoi on obtient ainsi une certaine optimalité. C'est la voie que je veux prendre et exposer à la fin du chapitre (section 8.7). Le lecteur intéressé y découvrira une notion de dérivations saturées, pour lesquelles la sémantique opérationnelle coïncide parfaitement avec la sémantique équationnelle.

8.3 Séquences & Champs d'Observation

Dans cette section, je donne une solution théorique au problème du calcul des stratégies. On peut voir ce calcul en deux temps. A supposer que l'on dispose d'un champs d'observation *a priori* $\mathcal{O}bs_1$, on peut dans un premier temps calculer des séquences pour chaque constructeur c. Ces séquences utiliseront des pas élémentaires de la forme $x = t$ avec $c \to x = t \in \mathcal{P}rg$ ou bien de la forme (P, R, x) avec $(P, R) \in \mathcal{O}bs_1$. Pour chacune de ces séquences, on peut identifier des paires (P, R) qui correspondent aux attributs paramètres et résultats nécessaires et produits par la séquence.

Donc, dans un second temps, en considérant l'ensemble de ces séquences obtenues, on peut détecter un nouveau champs d'observation *a posteriori* $\mathcal{O}bs_2$.

L'idée est alors de chercher le plus grand champs tel que $\mathcal{O}bs_1 = \mathcal{O}bs_2$. On aura donc une définition par point fixe de ce champs d'observation maximal. Dans cette section, je donnerai une description théorique de point fixe. Malheureusement, cette définition sera complètement inutilisable en pratique, mais dans les section 8.5 et 8.6 je donnerai des solutions plus techniques pour calculer des ensembles de séquences et le champs d'observation maximal.

8.3.1 Calcul des Séquences

Dans cette section, il est question de trouver des séquences en fonction d'un champs d'observation $\mathcal{O}bs$. En fait, il est naturel de ne chercher des séquences vérifiant quelques propriétés intuitives, limitant ainsi énormément la complexité du problème. Mais ce faisant, on risque de diminuer la sémantique $[\![\mathcal{P}rg]\!]^{op}$, ce qui n'est pas un problème en pratique (Cf. "première réaction" de la section 8.2).

Rappelons quelques faits. Une séquence est une suite de pas élémentaires notés $s_1, \ldots, s_n$. Chaque pas est ou bien l'application d'une équation $x = t$ sur la variable α, ou bien une dérivation récursive permettant de calculer des variables $x.r$ pour $r \in \mathcal{R}es$ en fonction de variables de la forme $x.p$ avec $p \in \mathcal{P}rm$.

Les séquences que je vais rechercher on de bonnes propriétés. Elles correspondent à des dérivations saturées, c'est-à-dire que une équation n'est pas appliquée si toutes les variables de sa partie droite ne sont pas définies auparavant. Dans ces conditions, je peux introduire les deux fonctions suivantes, définie pour tout pas, et indiquant les variables nécessaires et produite par un pas élémentaire:

$$\begin{array}{lcl} need(x = t) & = & \mathcal{V}ars(t) \\ prod(x = t) & = & \{x\} \\ need(P, R, y) & = & \{y.p \mid p \in P\} \cup \{y\} \\ prod(P, R, y) & = & \{y.r \mid r \in R\} \end{array}$$

Informellement, si les variables $need(s)$ sont définies avant le pas s, alors les variables $prod(s)$ seront aussi définies après l'application du pas s. Les dérivations de type sous-terme, primitives et de substitution ne m'intéressent pas dans ce cadre.

Dès lors, si je me limite à rechercher des dérivations saturées, il suffit de construire des dérivations ayant la forme suivante:

$$\Sigma_1 \xrightarrow{s_1} \Sigma_2 \xrightarrow{s_2} \ldots \Sigma_{n+1}$$

Si la séquence $[s_1; \ldots; s_n]$ est candidate pour une dérivation saturée du type $\overset{P,R,\alpha}{\longrightarrow}_{\mathcal{S}}$, on doit respecter certaines contraintes. Ainsi, en notant D_i l'ensemble des variables définies dans le système Σ_i, on doit avoir pour tout i:

$$\left\{ \begin{array}{c} prod(s_i) \subset D_{i+1} \\ need(s_i) \subset D_i \end{array} \right\}$$

Bien évidemment, on ne peut pas utiliser le couple (P, R) qui correspondra à la séquence, puisque c'est ce que l'on veut calculer dans un deuxième temps. Par ailleurs, comme on ne s'intéresse pas aux dérivations de type sous-terme ou substitution, on peut considérer que les variables α et $\alpha.k$ sont définies dès le début de la séquence. Pour ne pas avoir de contraintes liée à la paire (P, R), il suffit d'y ajouter les variables $\alpha.p$ avec $p \in \mathcal{P}rm$ quelconque. Les contraintes ci-dessus sont alors facilement simplifiée par la définition ci-dessous:

Définition 8.3.1 (Séquence saturée)
Une séquence $[s_1; \ldots; s_n]$ est saturée si et seulement si on a:

$$\left\{ \begin{array}{l} D_0 = \{\alpha\} \cup \{\alpha.k \mid k > 0\} \cup \{\alpha.p \mid p \in \mathcal{P}rm\} \\ (\forall i) \quad need(s_i) \subset D_0 \cup \bigcup_{j<i} prod(s_j) \end{array} \right.$$

Cette propriété est notée $Sat([s_1 \ldots; s_n])$.

Je rappelle que j'ai un champs d'observation *a priori* pour limiter l'espace de recherche des séquences, et que je cherche des séquences pour chaque constructeur de manière séparée. Je ne vais donc considérer que l'ensemble de pas élémentaires $S_c(\mathcal{O}bs)$ défini par:

$$\begin{array}{ll} S_c(\mathcal{O}bs) & = \{x = t \mid c \to x = t \in \mathcal{P}rg\} \\ & \cup \{P, R, y \mid (P, R) \in \mathcal{O}bs, y \in \mathcal{X}_{loc}\} \end{array}$$

J'ai maintenant tout ce qu'il me faut pour définir la fonction $Seq_(\mathcal{O}bs)$.

Définition 8.3.2 (Séquences)
L'ensemble des séquences possibles avec un champs d'observation a priori $\mathcal{O}bs$ *et pour le constructeur c est:*

$$Seq_c(\mathcal{O}bs) = \{seq \in S_c(\mathcal{O}bs)^* \mid Sat(seq)\}$$

Maintenant, je peux passer au deuxième temps qui calcule à partir de ces séquences un nouveau champs d'observation.

8.3.2 Calcul du Champs d'Observation

A partir d'une séquence seq, je peux définir un ensemble de paires (P, R) telles que, si $\Sigma \in \mathcal{I}nit(P, R)$ et $\Sigma' \in \mathcal{F}inal(P, R)$, alors on aura $\Sigma \stackrel{seq}{\to} \Sigma'$. En effet, il suffit pour cela de glaner les attributs p et r tels que dans au moins un pas de la séquence, $\alpha.p$ ou $\alpha.r$ est utilisé ou produit. A cet effet, j'introduis les fonctions suivantes:

Définition 8.3.3 (Paramètres et Résultats d'une séquences)
Étant donné une séquence seq, je définis:

$$\begin{cases} prm(seq) = \{p \in \mathcal{P}rm \mid (\exists s \in seq)\ \alpha.p \in need(s)\} \\ res(seq) = \{r \in \mathcal{R}es \mid (\exists s \in seq)\ \alpha.r \in prod(s)\} \end{cases}$$

Les ensembles d'attributs $prm(seq)$ et $res(seq)$ représente les contraintes minimales qu'il faut respecter pour que (P, R) soit une paire convenable. Maintenant, on peut aussi se contenter de moins de résultats, ou de plus de paramètres. Je définis donc l'ensemble des paires (P, R) qui conviennent pour la séquence seq, ensemble que j'appelle son profil:

Définition 8.3.4 (Profil d'une séquence)
Le profile de la séquence seq est défini par:

$$prof(seq) = \{(P, R) \mid P \supset prm(seq) \quad R \subset res(seq)\}$$

Pour qu'une paire (P, R) puisse faire partie d'un champs d'observation, il faut que pour tout constructeur, il existe une séquence dont le profile contienne la paire (P, R). Plus exactement, pour tous les constructeurs impliqués dans le calcul des attributs R. Je vais donc utiliser à nouveau un ensemble que l'on a déjà rencontré:

$$\tau_R = \{c \mid (\forall r \in R)(\exists t)\ c \to \alpha.r = t \in \mathcal{P}rg\}$$

Définition 8.3.5 (Calcul du Champs)
Je définis comme suit la fonction F, qui calcule un champs d'observation a posteriori *à partir des séquences issues d'un champs d'observation* a priori*:*

$$F(\mathcal{O}bs) = \{ \quad (P, R) \mid (\forall c \in \tau_R)\ (\exists seq \in Seq_c(\mathcal{O}bs))\ (P, R) \in prof(seq) \quad \}$$

Pour que le champs $\mathcal{O}bs$ soit convenable, il faut et il suffit que l'on ait $F(\mathcal{O}bs) = \mathcal{O}bs$. Ainsi, les stratégies qui fonctionnent pour les dérivations sur les variables de $\mathcal{X}_{loc}$ fonctionnent également sur α. Il ne reste plus qu'à calculer le plus grand point fixe de F.

Définition 8.3.6 (Champs d'Observation maximal)

Le champs $\mathcal{O}bs_\infty$ maximal est défini par $\mathcal{O}bs = \text{gfix } F$.

A partir de ce champs maximal, il est alors facile de retrouver toutes les stratégies. Il suffit en effet de collecter les séquences obtenues en calculant $Seq_c(\mathcal{O}bs_\infty)$, et de les affubler des paires (P, R) qui sont dans le profile de ces séquences. Je donne donc la définition suivante:

Définition 8.3.7 (Stratégies d'un programme)

L'ensemble des stratégies $\mathcal{S}$ du programme $\mathcal{P}rg$ est obtenu à partir de son champs d'observation maximal $\mathcal{O}bs_\infty$ de la manière suivante:

$$\mathcal{S} = \{(P, R, c, seq) \mid seq \in Seq_c(\mathcal{O}bs_\infty) \quad (P, R) \in prof(seq)\}$$

En pratique, ces définitions théoriques ne suffisent pas, car les séquences sont difficiles à calculer (il y en a beaucoup trop) et le point fixe de la fonction F également (les champs d'observation sont trop gros aussi). Pourtant, il est nécessaire de calculer les stratégies d'un programme, ou plus exactement, des stratégies d'un programme. En effet, que ce soit pour les transformations de réduction (Cf. section 5.8) ou pour la traduction d'un programme équationnel en un programme fonctionnel (Cf. chapitre 9), les stratégies jouent un rôle crucial.

Je me propose maintenant dans la fin de ce chapitre de développer ces deux points. La section 8.5 traite de la manière de calculer les séquences, en présentant trois affinements successifs de la fonction Seq_c, le dernier s'avérant en pratique le seul utilisable. Ensuite, la section 8.6 modifie en profondeur la fonction F pour représenter de manière astucieuse les champs d'observation à l'aide de base d'ensembles coniques, un outil algébrique qui est exposé à la section 10.5.

8.4 Exemple

Je reprends l'exemple de la section 8.1 pour illustrer comment les définitions précédentes permettent de retrouver les stratégies associées au programme de l'inversion de liste. Je rappelle ce programme et la numérotation de ces équations:

$$\left\{ \begin{array}{ccll} cons & \rightarrow & \alpha.r = \alpha.2.r & (\pi_1) \\ cons & \rightarrow & \alpha.2.p = (cons\ \alpha.1\ \alpha.p) & (\pi_2) \\ nil & \rightarrow & \alpha.r = \alpha.p & (\pi_3) \end{array} \right\}$$

Il n'y a que deux paires possibles pour composer le champs d'observation: $(\emptyset, r)$ et (p, r) (Pour simplifier, je ne mentionne pas les paires pathologiques qui ne produisent aucun résultat). Reste à trouver quel est le plus grand point fixe. Je vais démontrer que seule la paire $(\emptyset, r)$ ne peut pas appartenir au champs maximal.

En effet, prenons un champs $\mathcal{O}bs$ *a priori* qui contienne cette paire $(\emptyset, r)$, et voyons les séquences que l'on peut obtenir:

$$\begin{array}{l} Seq_{cons}(\mathcal{O}bs) = \{[(\emptyset, r, \alpha.2); \pi_1] \ldots\} \\ Seq_{nil}(\mathcal{O}bs) = \{[\pi_3]\} \end{array}$$

Alors la paire $(\emptyset, r)$ ne peut pas appartenir à $F(\mathcal{O}bs)$. En effet, on a bien:

$$prof([(\emptyset, r, \alpha.2); \pi_1]) = \{(\emptyset, r)\ (p, r)\}$$

Mais on a pour l'autre constructeur:

$$prof([\pi_3]) = \{(p, r)\}$$

Donc la paire $(\emptyset, r)$ n'appartient pas aux profils des séquences pour les deux constructeurs de $\tau_r = \{cons \quad nil\}$. On peut alors vérifier que, en revanche, (p, r) appartient au point fixe. On a alors:

$$\begin{array}{l} Seq_{cons}(p,r) = \{[\pi_2; (p,r,\alpha.2); \pi_1]\} \\ Seq_{nil}(p,r) = \{[\pi_3]\} \end{array}$$

Pour les profils on a alors:

$$\begin{array}{l} prof([\pi_2; (p,r,\alpha.2); \pi_1]) = \{(p,r)\} \\ prof([\pi_3]) = \{(p,r)\} \end{array}$$

On a donc $\mathcal{O}bs_\infty = \{(p,r)\}$ et l'ensemble des stratégies est finalement:

$$\mathcal{S} = \left\{ \begin{array}{l} (p,r,cons,[\pi_2; (p,r,\alpha.2); \pi_1]) \\ (p,r,nil,[\pi_3]) \end{array} \right\}$$

On retrouve bien les résultats de la section 8.1.

8.5 Calcul des Séquences

Donc, étant donné ce fameux champs d'observation $\mathcal{O}bs$ et un constructeur c, je peux calculer des séquences correspondant au contraintes précédentes par un algorithme de point de fixe qui construit à la i-ième itération le pas s_i et l'ensemble X_i. L'étape suivante, section 8.3.2, consistera bien entendu à trouver ce champs $\mathcal{O}bs$ et à réunir les séquences obtenues pour chaque constructeur.

8.5.1 Première version

Plus exactement, je calcule un ensemble de séquences possible. Je définis donc la fonction $Seq_c(\mathcal{O}bs)$ qui calcule les couples (seq, X), seq étant une séquence et X étant l'ensemble X_i associé au dernier pas de la séquence. Le calcul se fait par point fixe en itérant la fonction $AddStep$ qui ajoute à un ensemble de séquences un pas $s \in S_c$ si c'est possible.

$$\begin{array}{lcl} Seq_c(\mathcal{O}bs) & = & \text{fix } AddStep(\mathcal{O}bs)\ \{([], \{\alpha\})\} \\ AddStep(\mathcal{O}bs)(E) & = & E \\ & \cup & \{([seq; s], X \cup prod(s)) \mid s \in S_c(\mathcal{O}bs), need(s) \subset X\} \end{array}$$

Mais il faut ensuite déterminer les paires (P, R) associées aux paires (seq, X) ainsi calculées. Je propose en fait de le faire en parallèle, dans une deuxième version de l'algorithme.

8.5.2 Deuxième version

Ici, je calcule directement un ensemble de triplet (P, R, seq), chacun correspondant à une séquence recherchée avec les ensembles P et R qui conviennent. J'en profite pour faire une petite optimisation en évitant qu'une séquence produise deux fois la même variable:

$$
\begin{array}{ll}
Seq_c(\mathcal{O}bs) & = f(\text{fix } AddStep_{\mathcal{O}bs}\ \{(\emptyset, \emptyset, [], \{alpha\})\}) \\
 & \text{avec} \quad f(E) = \{(a,b,c) \mid (\exists d)\ (a,b,c,d) = (a,b,c)\} \\
AddStep_{\mathcal{O}bs}(E) & = E \cup \{add(s,e) \mid s \in Y_c(\mathcal{O}bs), verify(s,e)\} \\
add(s,(P,R,seq,X)) & = (P \cup P', R \cup R', [seq; s], X \cup prod(s)) \\
 & \text{avec} \quad P' = \{p \mid \alpha.p \in need(s)\} \\
 & \text{et} \quad R' = \{r \mid \alpha.r \in prod(s)\} \\
verify(s,(P,R,seq,X)) & = need(s) \subset X \text{ et } prod(s) \cap X = \emptyset
\end{array}
$$

Cet algorithme termine de manière évidente, car la taille des séquences calculées augmente strictement à chaque étape du calcul. On verra cependant que l'on calcule encore beaucoup trop de séquences.

8.5.3 Troisième version

Mes premiers essais de calcul de $Seq_c(\mathcal{O}bs)$ donnèrent des résultats absolument catastrophiques, d'une complexité épouvantable. Regardant d'un peu plus près, je me suis vite aperçu que le problème venait essentiellement de la fonction $AddStep_c$, qui calcule bien trop de mauvaises séquences. J'entends par là des séquences qui calculent de manière très redondante certaines paires (P, R), alors qu'une seule aurait suffit. Il y a donc un gros problème dans cette fonction, qui provient de deux origines très différentes:

- la plupart des séquences générées sont beaucoup trop longues par rapport au nombre de variables qu'elles peuvent définir. Ce phénomène est illustré par le fait que toutes les séquences associées à une paire (P, R) on toutes leurs pas élémentaires dans des séquences associées à des paires (P, R') avec $R' \subset R$.
- par ailleurs, de nombreuses séquences sont identiques modulo une permutation de l'ordre des pas élémentaires. Bien entendu, toute permutation n'est pas valide, mais hélas, le peu qui le sont introduisent une abondante redondance dans les séquences générées.

Ces deux remarques permettent d'élaborer le scénario de calcul suivant, qui est celui que j'utilise en pratique et qui donne d'excellents résultats:

- Redéfinir la fonction Seq_c de manière à ce qu'elle ne calcule qu'un ensemble R fixé à l'avance. Donc on aura $(P, seq) \in Seq_c(\mathcal{O}bs, R)$ si seq est une séquence correspondant à la paire (P, R).
- Générer les pas élémentaires par paquets, donc à une permutation près, quitte à vérifier *a posteriori* qu'il est possible de les ordonner correctement.

- Ne pas générer plus de pas que nécessaire, donc toujours diriger la production des séquences par un ensemble de variables dites nécessaires.

Pour générer les séquences, le noyau de l'algorithme est constitué d'une file d'état. Chaque état est un quadruplet (P, D, N, seq) dont la signification est la suivante: la séquence seq nécessite la définition initiale de l'ensemble de variables $\{\alpha.p \mid p \in P\}$, elle produit alors l'ensemble des variables D; N est l'ensemble des variables qu'il reste à définir en augmentant la séquence seq.

A chaque itération, on prend un état $E = (P, D, N, seq)$ dans la file et de deux choses, l'une: ou bien l'ensemble des variables restant à définir pour cet état est non vide ($N \neq \emptyset$) et on cherche alors à augmenter la séquence, ou bien il est vide ($N = \emptyset$) et on a une séquence complète. Dans le premier cas on génère de nouveaux états que l'on met dans la file, dans le second cas, on rajoute la séquence obtenue au résultat de la fonction Seq_c. Initialement, l'état de départ est réduit à $(\emptyset, \emptyset, \{\alpha.r \mid r \in R\}, [])$.

Le problème est maintenant de générer des pas "en paquet" pour définir les variables manquantes. Essentiellement, le problème se réduit à trouver tous les pas dans $S_c(\mathcal{O}bs)$ qui produisent des variables de N. Cependant, il ne faut pas que deux pas génèrent deux fois la même valeur. Le problème s'énonce donc ainsi: il faut trouver tous les paquets de plus grande taille possible qui génèrent des variables dans N, et tels que chaque variable n'est définie qu'une seule fois. C'est ici que l'algorithme de recherche des parties compatibles maximales, nommé *maxcompat* et présenté à la section 10.6, va intervenir.

Ensuite, il restera à filtrer les séquences obtenues, en vérifiant que l'on peut les ordonner de manière à respecter les dépendances entre variables nécessaires et produites par chaque pas élémentaire. Ce n'est pas difficile, grâce à un graphe établis à partir des relations *need* et *prod* et d'un algorithme de tri topologique (voir section 10.7).

$Seq_c(\mathcal{O}bs, R)$	calcule un ensemble de couple (P, seq):
1.	calculer $S_c(\mathcal{O}bs)$;
2.	initialiser la file F avec l'état initial $E = (\emptyset, \emptyset, \{\alpha.r \mid r \in R\}, [])$.
3.	tant que la file n'est pas vide.
3.1	prendre un état E dans F.
3.2	calculer $NextStates(E)$.
3.3	accumuler les nouveaux états renvoyés dans la file F, et accumuler les couples (P, seq) trouvés au cours du point 3.2.
4.	renvoyer les résultats accumulés.

$NextSteps(E)$	calcule des embryons de résultat et des nouveaux états. Posons $E = (P, D, N, seq)$:
•	si $N = \emptyset$:
1.	essayer d'ordonner la séquence seq et récupérer $seq' = Ord(seq)$.
2.	si c'est possible, on a trouvé un couple (P, seq'), sinon, on abandonne l'état E.
•	si $N \neq \emptyset$:
1.	calculer $S_0 = \{s \in S_c(\mathcal{O}bs) \mid prod(s) \subset N\}$.
2.	calculer $ES = maxcompat\ compat\ S_0$, avec la fonction $compat$ définie ci-après.
3	pour chaque paquet de pas $S \in ES$, renvoyer un nouvel état $AddSteps(E, S)$.
$compat\ s_1\ s_2$	$\Leftrightarrow prod(s_1) \cap prod(s_2) = \emptyset$.

$AddSteps(E, S)$	ajoute le paquet de pas S à l'état E. Posons $E = (P, D, N, seq)$:
1	calculer: $Prod = \bigcup_{s \in S} prod(s)$ $Need = \bigcup_{s \in S} need(s)$
2	renvoyer $(P', D', N', seq \cup S)$ avec: $P' = P \cup \{p \mid p \in \mathcal{P}rm, \alpha.p \in Need\}$ $D' = D \cup Prod$ $N' = (N - Prod) \cup Need$

Pour ordonner les séquences, on se sert naturellement d'un tri topologique de graphe (Cf. section 10.7):

$Ord(s_1; \ldots; s_n)$	ordonne la séquence.
1.	G est initialement un graphe vide.
2.	pour tout i, ajouter les arcs $x \to y$ dans G avec $x \in need(s_i)$ et $y \in prod(s_i)$.
3.	si le tri topologique de G est possible, ordonner les nœuds du graphe et renvoyer la séquence ainsi formée.

Sous cette forme, on sait donc efficacement générer des séquences à condition de connaître *a priori* un champs d'observation $\mathcal{O}bs$. Encore faut-il le calculer, ce qui n'est pas non plus un problème simple.

8.6 Calcul du Champs d'Observation

Je rappelle la définition de la fonction F (Cf section 8.3.2) qui calcule un champs d'observation *a posteriori* en fonction d'un champs *a priori*:

$$F(\mathcal{O}bs) = \{ \quad (P,R) \mid (\forall c \in \tau_R)\ (\exists seq \in Seq_c(\mathcal{O}bs))\ (P,R) \in prof(seq) \quad \}$$

Cette fonction F est difficile à calculer si on la présente ainsi. Une autre façon d'écrire cette définition est la suivante, dans laquelle on a regroupé la notion de profile et d'existence de séquence dans une fonction annexe notée $Field_c$. De plus, j'utilise la fonction Seq définie à la section 8.5 qui prends en paramètre la composante R d'une paire (P,R) (Cf. troisième version). On obtient ainsi:

$$Field_c(\mathcal{O}bs, R) = \{(P,R) \mid (\exists P' \subset P)(P', seq) \in Seq_c(\mathcal{O}bs, R)\}$$
$$F(\mathcal{O}bs) = \bigcup_{(P,R)\in\mathcal{O}bs} \bigcap_{c\in\tau_R} Field_c(\mathcal{O}bs, R)$$

On peut vérifier alors que F est monotone avec l'inclusion ensembliste, ce qui assure entre autre l'existence d'un point fixe. En utilisant les résultats de la section 10.2 concernant les calculs de points fixes et en notant $\mathcal{O}bs_0$ le plus grand champs d'observation possible (par rapport à l'inclusion ensembliste), on obtient alors:

$$\mathcal{O}bs_\infty = \text{fix } F\ \mathcal{O}bs_0$$

Malheureusement, avec cette méthode, il faut prendre pour $\mathcal{O}bs_0$ le plus grand élément du domaine. Ici, l'ordre considéré est l'inclusion ensembliste, donc D_0 est l'ensemble de toutes les paires possibles:

$$\mathcal{O}bs_0 = \{(P,R) \mid P \subset \mathcal{P}rm, R \subset \mathcal{R}es\}$$

Cet ensemble est vraiment beaucoup trop gros pour être utilisé en pratique vue le nombre astronomique de séquences engendrées par les gros champs d'observation. Ce n'est donc pas du tout la bonne méthode. Je vais donc essayer de trouver une autre fonction que F. L'ordre considéré ne sera plus le même, et il sera remplacé par celui définit au début cette section, pour lequel les champs d'observation sont coniques (Cf. section 10.5).

8.6.1 Représentation par des cônes

Le problème pour calculer le point fixe définissant $\mathcal{O}bs_\infty$ est qu'il porte sur des ensembles qui, bien qu'étant finis sont néanmoins très gros. En pratique, le calcul exhaustif n'est pas envisageable même en se limitant à des cas simples. Je vais donc devoir réduire les calculs à effectuer. C'est ici que je me servirai de la section 10.5 relative aux parties coniques. Je recommande donc au lecteur de s'y reporter dès maintenant, ne serait-ce que pour se familiariser aux notations. Ceci fait, je considère la relation d'ordre définie plus haut, que je rappelle ici:

$$(P,R) > (P',R') \Leftrightarrow P' \subset P \text{ et } R' = R$$

Or, F vérifie une très bonne propriété qui est: si A est conique, alors $F(A)$ aussi. Donc, $\mathcal{O}bs_0$ étant conique et avec $\mathcal{O}bs_m = F^m(\mathcal{O}bs_0)$, on prouve par récurrence que $\mathcal{O}bs_m$ est conique pour tout m et par continuité que $\mathcal{O}bs_\infty$ l'est également. On va donc se contenter de calculer la suite des bases de ces champs. Soit donc $A_m = base\ \mathcal{O}bs_m$ pour tout m et je rappelle que l'on a $\mathcal{O}bs_m = A_m^\uparrow$. Donc je peux en déduire les relations suivantes:

$$\begin{array}{l} A_0 = base\ \mathcal{O}bs_0 \\ A_{m+1} = G(A_m) \\ \text{avec} \quad (\forall A)\ G(A) = base\ F(A^\uparrow) \end{array}$$

Pourquoi cette idée donne-t-elle de bons résultat? Tout vient de la façon dont on calcule les séquences. En bref, la fonction *Field* est presque insensible au cônes car la fonction *Seq* vérifie une bonne propriété de monotonie. Le lecteur pourra vérifier que l'on a en effet:

$$(P, seq) \in Seq_c(A^\uparrow, R) \Rightarrow (\exists seq', \exists P' \subset P)\ (P', seq') \in Seq_c(A, R)$$

C'est un résultat un peu technique à démontrer, mais sans difficulté. Intuitivement, il est directement lié à la notion de production et de nécessités et au deuxième problème vu dans la section 8.3.2. La clef de la démonstration consiste à remplacer les pas élémentaires de la forme (P, R, y) avec $(P, R) \in A^\uparrow$ par un autre pas. Or, comme $(P, R) \in A^\uparrow$, il existe $(P', R) \in A$ tel que $(P', R) < (P, R)$. On vérifie alors que l'on peut remplacer le pas (P, R, y) par (P', R, y) sans trop modifier le reste de la séquence.

Un corollaire intéressant concerne la fonction $Field_c$, pour laquelle on peut écrire:

$$base\ Field_c(A^\uparrow, R) = base\ Field_c(A, R)$$

C'est un résultat obtenu par application directe de la propriété précédente et par monotonie par rapport à A de la fonction $Field_c(A, R)$.

8.6.2 Point fixe, première mouture

Les calculs qui suivent sont pas entièrement détaillés. Je me contente ici de donner les grandes étapes d'un long calcul qui permet d'aboutir à une autre écriture du point fixe qui soit calculable en pratique. La fonction que je veux calculer étant $G(A) = base\ F(A^\uparrow)$, un premier jeu d'écriture permet d'arriver à la formule suivante:

$$F(A^\uparrow) = \bigcup_{(P,R)\in A^\uparrow} \bigcap_{c\in\tau_R} Field_c(A, R)^\uparrow$$

Or, pour toute paire $(P, R) \in A^\uparrow$, la condition $c \in \tau_R$ est équivalente à $c \in \tau_R$ avec $(P, R) \in A$ et de plus, P n'intervient pas ailleurs dans le calcul. On peut donc réduire le domaine de l'union à $(P, R) \in A$ au lieu de $(P, R) \in A^\uparrow$. Ce qui donne:

$$F(A^{\uparrow}) = \bigcup_{(P,R)\in A} \left(\bigcap_{c\in\tau_R} \mathit{Field}_c(A,R)^{\uparrow} \right)$$

Si je reprends la définition de la fonction G, que j'utilise les théorèmes de la section 10.5 ainsi que les relations de monotonies vues plus haut, j'obtiens finalement la formule suivante:

$$G(A) = \mathit{base} \bigcup_{(P,R)\in A} \left(\bigwedge_{c\in\tau_R} \mathit{Field}_c(A,R) \right)$$

Cette formule est déjà plus facile à calculer que les précédentes, surtout grâce aux algorithmes de calcul de $\wedge$ et de la fonction *base* décris à la section 10.5.

Je n'ai pas encore vraiment gagné, car il faut vérifier que les ensembles A_m restent suffisamment petits pour que ce soit intéressant. Or A_0 est tout de même assez gros, à savoir le plus grand champs d'observation possible par rapport à la relation d'ordre $<$ définie sur les paires (P,R). Donc:

$$A_0 = \{(\emptyset, R) \mid R \subset \mathcal{R}es\}$$

En fait, on aurait pu choisir un ordre différent pour les champs, en rendant symétrique le comportement des attributs résultats et paramètres. Prenons l'ordre défini ainsi:

$$(P,R) >_2 (P',R') \Leftrightarrow P \supset P' \text{ et } R \subset R'$$

On obtient alors presque la même formule pour G, $\mathit{Field}_c(A,R)$ étant remplacé par $\mathit{Field}_c(A)$. L'avantage serait alors que maintenant le point de départ se réduit à $A_0 = \{(\emptyset, \mathcal{R}es)\}$. On ne pourrait espérer mieux! Sauf en remarquant que pour obtenir ces résultats avec l'ordre $>_2$ à la place de $>$, la troisième version de la fonction Seq_c n'est pas utilisable et qu'il faut se rabattre sur la seconde. C'est alors très embêtant car on a vu que cette deuxième version est trop longue à calculer.

En pratique, je me suis donc aperçu que le premier ordre $>$ était une option bien plus efficace que $>_2$. Évidemment, il est impossible de commencer le calcul du point fixe avec $A_0 = \{(\emptyset, R) \mid R \subset \mathcal{R}es\}$. Donc, il va falloir ruser un peu.

8.6.3 Point fixe, deuxième mouture

Pour produire une séquence correspondant à la paire (P,R), on peut remarquer qu'il suffirait de mélanger les séquences associées aux paires (P,r) avec $r \in R$. Cette remarque laisse penser que, dans un premier temps, on peut se contenter du point fixe suivant:

$$\begin{array}{rcl} B_0^0 & = & \{(\emptyset, r) \mid r \in \mathcal{R}es\} \\ B_\infty^0 & = & \text{fix } G\ B_0^0 \end{array}$$

Alors, le champs d'observation B^0_∞ génère (au sens des cônes) toutes les paires de la forme (P, r) qui font partie du champs d'observation recherché, $\mathcal{O}bs_\infty$. Maintenant, considérons les séquences obtenues de cette manière. Pour chacune d'entre elle, on peut chercher tous les attributs résultats qui sont calculés sur une même variable. On peut alors relancer le point fixe en cherchant des paires qui regroupent tout ou partie de ces attributs résultats.

Prenons un exemple: si dans une séquence on a les deux pas élémentaires (P_1, R_1, y) et (P_2, R_2, y), donc sur la même variable y, on peut chercher des paires de la forme (P, R) avec $R \subset R_1 \cup R_2$ pour essayer de regrouper des calculs. Je vais donc chercher une fonction notée *tupling* qui calcule des candidats R au regroupement en fonction d'un champs d'observation donné. Ainsi, on aura une suite de points fixes, qui convergera vers un dernier point fixe, que l'on espère aussi proche que possible de la base de $\mathcal{O}bs_\infty$.

$$\begin{array}{rcl} C_0 & = & \{(\emptyset, r) \mid r \in \mathcal{R}es\} \\ C_\infty & = & \text{fix } H \ C_0 \\ H(X) & = & \textit{tupling} \ (\text{fix } G \ X) \end{array}$$

La corrélation entre C_∞ et la base de $\mathcal{O}bs_\infty$ dépend essentiellement de la quantité de paires que génère la fonction *tupling*. Or, pour limiter le coût de la recherche, on a intérêt à en générer le moins possible.

Car en effet, pour bon nombre des paires que l'on pourrait générer, la totalités des séquences associées ne sont rien d'autre qu'une juxtaposition de séquences déjà connues. On peut donc considérer que connaître des paires (P_i, R_i) suffit pour générer la paire $(\bigcup P_i, \bigcup R_i)$.

On a donc intérêt à ne générer dans la fonction *tupling* que des paires qui permettent en effet de mettre en commun des pas élémentaires identiques, donnant lieu à de nouvelles séquences.

Je ne me proposerai pas d'examiner profondément cet aspect des choses. Dans le cadre de cette thèse, je vais donc proposer deux fonctions *tupling*, l'une est exhaustive, donc coûteuse mais exacte, l'autre se base sur une considération simple, mais ne donne qu'un résultat approché.

Pour la méthode exhaustive, on cherche dans une même séquence deux pas de la forme (P_1, R_1, y) et (P_2, R_2, y) (avec la même variable y). Chaque fois que l'on trouve une telle situation, on génère toute les paires $(\emptyset, R)$ avec $R \subset R_1 \cup R_2$.

Pour la deuxième méthode, on collecte tout d'abord l'ensemble des paires (P_1, R_1) et (P_2, R_2) comme pour la première méthode, puis on collecte une autre information qui servira de filtre. On cherche donc dans une même séquence si il existe une dépendance entre deux variable $y.r$ et $y.r'$, et on relève tout les couples (r, r') considérés. L'idée du filtrage est alors la suivante: si on cherche à générer une paire $(\emptyset, R)$ avec R qui contiennent les attributs r et r' relevés précédemment, on est sûr de ne pas pouvoir vraiment tirer partie de leur regroupement, car $y.r$ et $y.r'$ doivent être calculés l'un après l'autre à cause de leur interdépendance.

Donc, au lieu de générer toutes les paires $(\emptyset, R)$ avec $R \subset R_1 \cup R_2$, on prendra pour R les sous-séquences compatibles maximales dans $R_1 \cup R_2$ en considérant que les éléments incompatibles sont justement les couples (r, r') récoltés. On aura ainsi filtré bon nombre de fusions des ensembles R_i qui ne donnent rien de neuf.

Cette partie pourrait être encore améliorée sur la base des travaux de `Fnc-2`, qui introduit d'autres filtres du même genre.

8.7 Origine des Stratégies

8.7.1 Dérivation Saturées

La dérivation $\rightarrow_{\mathcal{P}rg}$ autorise des manipulations très intéressantes sur les programmes. Mais d'un autre côté, son aspect non-déterministe peu poser des problèmes. Ainsi, un des problèmes est lié au moment où sont effectuées les substitutions. Or, à part dans des sémantiques dites paresseuses (*lazy* en anglais), une variable reçoit sa valeur au moment même où elle est déclarée. Donc, pour rechercher une sémantique opérationnelle, il conviendrait de se limiter à des dérivations dans lesquelles les substitutions de variable ne sont pas effectuées trop tardivement.

Une dérivation saturée, notée $\rightarrow^{sat}_{\mathcal{P}rg}$ est une sous-relation de $\Rightarrow_{\mathcal{P}rg}$ qui vérifie certaines contraintes. Ce dont j'ai besoin quand $\Sigma \rightarrow^{sat}_{\mathcal{P}rg}$ se résume aux propriétés suivantes:

- Σ et Σ' ne contiennent que des équations de la forme $x = v$ où v est une valeur.
- Exactement une équation de $\mathcal{P}rg$ est appliquée. Si cette équation est $c \rightarrow y = t$ et qu'elle est appliquée sur la variable x, alors Σ doit définir x et toutes les variables de $\mathcal{V}ars(t[x])$.
- Les variables définies par Σ' sont exactement les mêmes que celles de Σ, augmentées de la variable $y[x]$ avec les mêmes notations.

Par extension, j'appelle également dérivation saturée une suite de systèmes $(\Sigma_i)_{i \leq n}$ telle que $\Sigma_i \rightarrow^{sat}_{\mathcal{P}rg} \Sigma_{i+1}$. La sémantique équationnelle saturée, qui n'utilise que des dérivations saturée, est alors notée $[\![\mathcal{P}rg]\!]^{sat}$.

8.7.2 Décomposition

Maintenant, je vais présenter intuitivement les principaux théorèmes qui sont associés aux dérivations saturées. Je rappelle que l'on a pas besoin explicitement de ces théorèmes pour pouvoir appliquer et utiliser la sémantique équationnelle et les transformations de programmes associées. Je ne donnerai donc pas la preuve des deux théorèmes suivants, qui sont assez complexes, mais une idée générale. Je cite ces théorèmes uniquement pour signaler au lecteur que la différence entre dérivations saturées et dérivations quelconques n'est pas si grande qu'on pourrait le penser.

Théorème 8.7.1 (Semi-complétude)
Pour tout programme $\mathcal{P}rg$, il existe un champs d'observation $\mathcal{O}bs$ et une stratégie $\mathcal{S}$ tels que pour tout $(P, R) \in \mathcal{O}bs$, si Σ_P définit P et Σ_R définit R, alors :

$$\Sigma_P (\rightarrow^{sat}_{\mathcal{P}rg})^* \Sigma_R \quad \Rightarrow \quad \Sigma_P \overset{P,R,\alpha}{\rightarrow_{\mathcal{S}}} \Sigma_R$$

On a donc pour tout programme et dans son champs associé les relations suivantes:

$$[\![\mathcal{P}rg]\!] \supset [\![\mathcal{P}rg]\!]^{sat} = [\![\mathcal{P}rg]\!]^{op}$$

La clef de la démonstration réside dans une méthode de décomposition qui permet, grâce à des permutations judicieuses dans l'ordre d'application des équations, de prouver par récurrence le théorème suivant:

Théorème 8.7.2 (Décomposition)
Avec les mêmes notations que ci-dessus, si on a:

$$\Sigma_P (\rightarrow^{sat}_{\mathcal{P}rg})^* \Sigma_R$$

alors il existe une séquence de pas élémentaires seq et une stratégie $\mathcal{S}$ telle que:

$$\Sigma_P \overset{seq,\alpha}{\rightarrow_{\mathcal{S}}} \Sigma_R$$

Pour faire cette décomposition, on commence par utiliser un ordre partiel sur les variables: $x < y \Leftrightarrow (\exists z)\ y = z[x]$. Avec cet ordre, on peut remarquer que toute dérivation $\Rightarrow_{\mathcal{P}rg}$ qui applique une équation de programme sur x ne porte que sur des variables plus grande que x. Ainsi, on peut faire des "paquets" pour regrouper des dérivations $\Rightarrow_{\mathcal{P}rg}$, de telle sorte que deux dérivations successives portent toujours sur des variables comparables. On construit ainsi des "appels récursifs", chaque récursion étant associée à un paquet de variables plus petites qu'une certaine variable x. On associe alors à ce paquet une stratégie appliquée à la variable x.

De ce théorème, on déduit une manière de fabriquer le champs $\mathcal{O}bs$ maximal par un point fixe. En effet, il suffit de générer des séquences de pas élémentaires et de chercher un champs d'observation qui se retrouve pour toute dérivation.

Chapitre 9

Traductions de Programmes

Ce chapitre est dédié à la transformation d'un programme fonctionnel en son équivalent équationnel, et réciproquement. Si vous n'êtes pas un adepte de la programmation fonctionnelle, ce chapitre ne vous parlera pas trop. Encore que j'ai pris soin de faire ressortir certaines idées qui ne sont pas liée à la nature fonctionnelle du langage source.

Ce chapitre est composé de deux parties. La première moitié du chapitre concerne la traduction d'un programme fonctionnel en programme équationnel et la moitié suivante est consacrée à la traduction inverse.

9.1 Du fonctionnel à l'équationnel

Le cœur d'un langage fonctionnel est la construction $(e_1\ e_2)$ qui applique une expression e_2 sur une expression e_1, cette dernière étant sensée être une fonction. Le cœur de la traduction en programme équationnel devra donc correspondre à cet opération.

La traduction que je définis permet d'associer à toute expression e un terme de la sémantique équationnelle. Donc toutes les opérations de la sémantique fonctionnelle $(\rightarrow_\gamma)$ doivent se retrouver dans des équations sur des attributs.

Si e_i est représentée par un terme t_i, je décide que, avec $x = t_1$ et $x.arg = t_2$, on aura $x.call = (e_1\ e_2)$. J'utiliserai donc toujours le même attribut résultat *call* et le même attribut paramètre *arg* pour représenter le calcul de l'application de deux expressions.

Une fois ceci décidé, il ne reste plus qu'à écrire la traduction. Par ailleurs, les programmes équationnels ont à leur disposition moult transformations qui les simplifient, par évaluation partielle ou spécialisation, etc. Donc, on peut se *contenter* d'une transformation qui soit très basique, voir simpliste. On peut même se permettre d'avoir une première traduction qui soit très éloignée de celle que l'on aurait écrite à la main.

Dans les exemples que je donne, je montrerai souvent les résultats obtenus à la traduction, immédiatements suivis de leur forme partiellement évaluée pour que le lecteur puisse y comprendre quelque chose.

9.2 Traduction des programmes fonctionnels

Pour effectuer la traduction d'un programme fonctionnel en programme équationnel, je procède en deux étapes. La première ne traite que des déclarations de fonction, la seconde ne s'occupe que du corps de ces fonctions, donc des expressions fonctionnelles.

Pour obtenir une traduction des expressions, le principe que j'utilise est simple. Pour une expression e, je vais construire un système Σ tel que $\Sigma \Rightarrow_{\mathcal{P}rg} \Sigma'$ avec Σ' qui représente le résultat du calcul de e, et $\mathcal{P}rg$ sera bien entendu la traduction du programme fonctionnel complet.

Comme la sémantique fonctionnelle est dénotationnelle, on pourra faire une traduction qui le soit également. Je veux dire par là que la traduction d'une expression sera une combinaison des traductions de ses sous-expressions.

Si le calcul d'une expression e est représenté par Σ, alors on s'attend à ce que $\Sigma \Rightarrow_{\mathcal{P}rg} \{x = v\}$, avec v qui soit la de l'évaluation de l'expression e. Encore faut-il choisir la variable x, se mettre d'accord sur la traduction des valeurs et finalement vérifier que tout cela fonctionne bel et bien.

Mais de part la nature dénotationnelle de la traduction, on aura besoin de modifier cette variable x pour pouvoir dénoter les traductions des différentes sous-expressions d'une expression. Donc la traduction prendra en paramètre une variable x (équationnelle) qui dénotera la valeur de l'expression à traduire.

Par ailleurs, une expression fonctionnelle contient des variables (fonctionnelles), donc la traduction devra prendre en paramètre un environnement assignant à chaque variable (fonctionnelle) le terme équationnel correspondant. Plus exactement, cet environnement Γ associera à chaque variable fonctionnelle a une fonction f_a, telle que pour toute variable équationnelle x, $f_a(x)$ renvoie un système Σ tel que $\Sigma \Rightarrow_{\mathcal{P}rg} \{x = t\}$ où t est un terme représentant la valeur associée à la variable fonctionnelle a.

Pour les applications de fonctions, comme je l'ai dit dans la section 9.1, j'utilise les attributs *arg* et *call* pour représenter le calcul d'une application. Pour traduire $(e_1\ e_2)$, il suffit donc de traduire e_1 dans une variable locale y, et de traduire e_2 sur $y.arg$. Le résultat de l'application sera alors disponible sur $y.call$.

Définition: Je note $[\![e]\!](\Gamma, x)$ la traduction dans la variable (équationnelle) x de l'expression (fonctionnelle) e avec l'environnement Γ.

Définition: Je note $\mathtt{fun}[\![D]\!]$ la traduction de la déclaration de fonction D.

Définition: Je note Γ_D l'environnement permettant de représenter toutes les déclarations du programme.

Dans ce qui suit, la fonction *newloc* est supposée renvoyer à chaque appel une nouvelle variable locale $\alpha.L_m$ qui n'a encore jamais été utilisée.

L'environnement Γ_D est défini en fonction de chaque déclaration du programme fonctionnel. Ainsi:

- pour chaque déclaration de constante de la forme `let f = e` on a:

$$\Gamma_D(f)(x) = \{x = y.value; y = (f)\}$$

  avec $y = newloc\ ()$.
- pour chaque fonction forme `let f x... = ...` on a:

$$\Gamma_D(f)(x) = \{x = (f_0)\}$$

Pour traduire les déclarations elles-mêmes, j'utilise la fonction fun$[\![\ ]\!]$ qui est définie différemment selon chaque forme de déclaration. Le point commun est cependant la création des fermetures. Pour chaque fonction f, j'utilise des constructeurs f_k pour représenter la fermeture de f dont les k premiers arguments sont connus, comme dans le chapitre 2. Il y a donc une fonction commune qui fabrique le programme équationnel de la fermeture f_k:

- $Closure(f, k) = f_k \to \{\alpha.call = (f_{k+1}\ \alpha.1...\alpha.k\ \alpha.arg)\}$

Pour une déclaration de fonction d'arité nulle (donc une constante) on a:

- $\mathrm{fun}[\![\mathtt{let} f = e]\!] = f \to \Sigma \quad \text{avec} \quad \Sigma = [\![e]\!]\ \alpha.value, \Gamma_D$

Pour une déclaration de fonction simple, on a:

- $\mathrm{fun}[\![\mathtt{let} f\ x_1...x_n = e]\!] = \bigcup_{0 \le k < n} Closure(f, k) \cup \{f_n \to \Sigma\}$
  avec :
  $\Sigma = [\![e]\!]\ \alpha.call, \Gamma$
  $\Gamma(x_k)(y) = \{y = \alpha.k\} \quad \text{pour} \quad 1 \le k < n$
  $\Gamma(x_n)(y) = \{y = \alpha.arg\}$
  $\Gamma = \Gamma_D \quad \text{sinon.}$

Pour une fonction définie par pattern-matching, la définition est un peu plus longue pour tenir compte de chaque pattern:

- $\mathrm{fun}[\![\mathtt{let}\ f\ x_1...x_n = \mathtt{func}\ y_1...y_m \to e_c \mid ...]\!] =$
  $(\bigcup_{0 \le k < n} Closure(f, k)) \cup \Pi \cup (\bigcup \Pi_c)$
  avec :
  $m = newloc\ ()$
  $\Pi = f_n \to \{$
  $\alpha.call = \alpha.L_m.f\_call$
  $\alpha.L_m.f\_x_k = \alpha.k \quad {}_{1 \le k \le n}$
  $\alpha.L_m = \alpha.arg$
  $\}$
  $(\forall c)\ \Pi_c = c \to [\![e_c]\!]\ \alpha.f\_call, \Gamma_c$

Les substitution Γ_0 et Γ_c étant définies ainsi:

$$\begin{array}{l}\Gamma_c(y_j)(z) = \{z = \alpha.j\} \\ \Gamma_c = \Gamma_0 \quad \text{sinon.} \\ \Gamma_0(x_k)(z) = \{z = \alpha.f\_x_k\} \\ \Gamma_0 = \Gamma_D \quad \text{sinon.}\end{array}$$

Pour les expressions, la fonction $[\![\ ]\!]$ est alors définie par:

$$\begin{array}{ll}\bullet[\![a]\!](\Gamma, x) & = \Gamma(a)(x) \\ \bullet[\![(e_1\ e_2)]\!](\Gamma, x) & = [\![e_1]\!](\Gamma, y) \\ & \cup[\![e_2]\!](\Gamma, y.arg) \\ & \cup\{x = y.call\} \\ & \text{avec} y = newloc() \\ \bullet[\![(\varphi\ e_1 \ldots e_n)]\!](\Gamma, x) & = \{x = (\varphi\ x_1 \ldots x_n)\} \\ & \cup \bigcup_{i \leq n}[\![e_i]\!](\Gamma, x_i) \\ & \text{avec}(\forall\varphi) \quad \varphi \in \mathcal{C}ons \cup \mathcal{P}rim \\ & \text{et}(\forall i \leq n) \quad x_i = newloc()\end{array}$$

9.3 Traduction des stratégies

Pour la traduction inverse, je rappelle pour éviter toute ambiguïté qu'elle n'est pas possible pour tout programme équationnel. Plus exactement, elle ne peut se faire qu'à partir des stratégies, voir chapitre 8, et sur un programme équationnel réduit, voir section 5.8.

La raison de toutes ces contraintes est finalement assez simple, et vient de la nécessité d'avoir un programme équationnel déterministe, au sens décrit en introduction dans la section 3.7.

9.3.1 Idée générale

Plus pratiquement, avec un tel programme équationnel $\mathcal{P}rg$, sa sémantique opérationnelle $[\![\mathcal{P}rg]\!]^{op}_{\mathcal{O}bs}$ est une stratégie $\mathcal{S}$ qui vérifie la propriétés suivante: pour tout constructeur c, pour toute paire $(P, R) \in \mathcal{O}bs$, il existe *au plus* une séquence seq telle que (P, R, c, seq) appartienne à $\mathcal{S}$.

Pour construire un programme fonctionnel qui corresponde à ces stratégies, l'idée est alors de construire, pour toute paire (P, R), une fonction `f` dont la définition sera de la forme:

```
let f p1 ... pn = fun
| c -> e
| ...
```

Sous cette forme, pour chaque pattern `c ->e` l'expression `e` construira le tuple des valeurs associées aux attributs résultats $\overline{R}$ de la stratégie $(P, R, \mathtt{c}, seq)$, tandis que les paramètres `p1`, ..., `pn` représente les valeurs associées aux attributs paramètres $\overline{P}$ de cette même stratégie.

Ce schéma général de traduction est très utilisé dans le contexte des grammaires attribuées. Il n'y a donc rien de révolutionnaire ici.

9.3.2 Ordre supérieur

Cependant, pour tenir compte de l'ordre supérieur, il est plus pertinent de tenir compte de quelques stratégies particulières. Dans le langage fonctionnel que j'ai choisi, il y a deux types de fonctions: celles qui sont définies "par cas" sur des constructeurs, et les autres que j'appelle fonctions "pures".

On peut retrouver facilement cette distinction dans les stratégies, et c'est un point intéressant pour pouvoir coder correctement des fonctions d'ordre supérieur. En effet, supposons que, pour le constructeur c, il n'existe qu'une seule paire (P, R) et donc qu'une seule séquence seq telle que (P, R, c, seq) soit définie. Alors, on peut dire que ce constructeur correspond à une fonction pure. Sinon, c'est que le constructeur correspond à un constructeur d'un certain type de donné et qu'il existe plusieurs fonctions qui doivent être définies "par cas" sur ce type de donnée.

Cette distinction est alors capturée par les définitions suivantes:

Définition: On définit les tables suivantes:

$$\begin{array}{lcl} D_{\mathcal{S}}[P,R,c] & = & seq \text{ si } (P,R,c,seq) \in \mathcal{S}\} \\ T_{\mathcal{S}}[P,R] & = & \{c \mid \exists seq \quad (P,R,c,seq) \in \mathcal{S}\} \\ C_{\mathcal{S}}[c] & = & \{(P,R) \mid \exists seq \quad (P,R,c,seq) \in \mathcal{S}\} \end{array}$$

Ces tables sont bien définies sous les conditions énoncées en tête de section. C'est notamment pour la définition de ces tables que les contraintes de déterminisme sont nécessaires.

Définition: $F_{\mathcal{S}}$ est l'ensemble des paires (P, R) pour lesquelles on a:

$$(P,R) \in F_{\mathcal{S}} \Leftrightarrow (\exists c)\ C_{\mathcal{S}}[c] = \{(P,R)\}$$

9.3.3 Définition des noms de fonctions

Deux catégories de noms de fonction sont définies, et qui correspondent aux fonctions "pures" et aux fonctions "par cas":

- Pour chaque paire (P, R) qui est dans $F_{\mathcal{S}}$, et pour chaque constructeur $c \in T_{\mathcal{S}}[P, R]$, un nouveau nom de fonction `f` est défini. On construit ainsi une table des fonctions "pures", notée $FP_{\mathcal{S}}$ et qui associe à tout constructeur c tel que $C_{\mathcal{S}}[c]$ est réduit à une seule paire (P, R) le triplet $(P, R, \mathtt{f})$ qui correspond.
- Pour chaque paire (P, R) qui n'est pas dans $F_{\mathcal{S}}$, on définit un nouveau nom de fonction `f`. On construit ainsi une table des fonctions "par cas", notée $FC_{\mathcal{S}}$ et qui associe à chaque paire (P, R) le nom de fonction `f` qui correspond.

9.3.4 Traduction des séquences

Le schéma de traduction d'une séquence utilise deux éléments: d'une par les définitions précédentes $F_{\mathcal{S}}$, $FP_{\mathcal{S}}$ et $FC_{\mathcal{S}}$, d'autre part un environnement Γ qui associe des variables *équationnelles* à des variables *fonctionnelles*. Je suppose l'existence d'une fonction `newvar()` qui crée à volonté des noms de variables fonctionnelles toutes différentes.

Tout d'abord, je définis la fonction $E[\![t]\!](\Gamma)$ qui calcule une expression associée au terme t dans l'environnement Γ:

$$\begin{array}{lll}
\bullet\quad E[\![x]\!](\Gamma) & = & \Gamma(x) \\
\bullet\quad E[\![(c\ t_1 \ldots t_n)]\!](\Gamma) & = & (c\ e_1 \ldots e_n) \\
 & \text{avec} & (\forall i)\ e_i = E[\![t_i]\!](\Gamma) \\
 & \text{et} & FP[c] = \bot \\
\bullet\quad E[\![(c\ t_1 \ldots t_n)]\!](\Gamma) & = & (\mathtt{f}\ e_1 \ldots e_n) \\
 & \text{avec} & (\forall i)\ e_i = E[\![t_i]\!](\Gamma) \\
 & \text{et} & FP[c] = (P, R, \mathtt{f}) \\
\bullet\quad E[\![(\pi\ t_1 \ldots t_n)]\!](\Gamma) & = & (c\ e_1 \ldots e_n) \\
 & \text{avec} & (\forall i)\ e_i = E[\![t_i]\!](\Gamma)
\end{array}$$

Ensuite, je défini la fonction $S[\![s]\!](\Gamma)$ qui calcule l'effet du pas élémentaire s sur l'environnement Γ. Cet effet se compose d'un nouvel environnement Γ' et d'une déclaration de la forme `let X = E`.

$$\bullet\quad S[\![x = t]\!] = (\Gamma[x{:=}y]\ ,\ \mathtt{let}\ y = e) \quad \text{avec} \begin{cases} e = E[\![t]\!](\Gamma) \\ y = \mathtt{newvar}() \end{cases}$$

$$\bullet\quad S[\![(P, R, x)]\!] = (\Gamma[x.r_1{:=}y_1 \ldots x.r_n{:=}y_n]\ ,\ \mathtt{let}\ (y_1, \ldots, y_n) = (\mathtt{f}\ z_1 \ldots z_m\ z))$$
$$\text{avec} \begin{cases} (P, R) \notin F_{\mathcal{S}}, \quad FC[P, R] = \mathtt{f}, \quad z = \Gamma(x) \\ (\forall i) y_i = \mathtt{newvar}(), \quad (\forall j) z_j = \Gamma(x.p_j) \\ \overline{P} = [p_1 \ldots p_m], \quad \overline{R} = [r_1 \ldots r_n] \end{cases}$$

$$\bullet\quad S[\![(P, R, x)]\!] = (\Gamma[x.r_1{:=}y_1 \ldots x.r_n{:=}y_n]\ ,\ \mathtt{let}\ (y_1, \ldots, y_n) = (z\ z_1 \ldots z_m))$$
$$\text{avec} \begin{cases} (P, R) \in F_{\mathcal{S}}, \quad z = \Gamma(x) \\ (\forall i) y_i = \mathtt{newvar}(), \quad (\forall j) z_j = \Gamma(x.p_j) \\ \overline{P} = [p_1 \ldots p_m], \quad \overline{R} = [r_1 \ldots r_n] \end{cases}$$

Pour toute séquence élémentaire je définis une fonction de compilation, notée *Compile*, qui renvoie une expression fonctionnelle associée à une séquence *seq* qui calcule les valeurs des variables $\alpha.r$ avec $r \in R$ dans l'environnement Γ:

$$\begin{array}{rl}
\bullet\, \mathit{Compile}\ R\ []\ \Gamma & = (x_1, \ldots, x_n) \\
\text{avec} & (\forall i)\ x_i = \Gamma(\alpha.r_i) \\
\text{et} & \overline{R} = [r_1 \ldots r_n] \\
\bullet\, \mathit{Compile}\ R\ [s; \mathit{seq}]\ \Gamma & = \text{"}\mathtt{let\ X = A\ in\ E}\text{"} \\
\text{avec} & (\Gamma', \mathtt{let\ X = A}) = S[\![s]\!](\Gamma) \\
\text{et} & \mathtt{E} = \mathit{Compile}\ R\ \mathit{seq}\ \Gamma'
\end{array}$$

9.3.5 Construction du programme

Le programme fonctionnel peut maintenant être construit complètement. Il est constitué de deux parties, l'une pour les fonctions "pures" et l'autre pour les fonctions "par cas":

- Pour chaque entrée dans la table $FP_{\mathcal{S}}$ telle que $FP_{\mathcal{S}}[c] = (P, R, \mathtt{f})$, je crée la déclaration de fonction suivante:

  ```
  f X1 ... Xn = E
  ```

  avec les définitions suivantes:

  $$\begin{array}{l} \overline{P} = \{p_1, \ldots, p_n\} \\ \mathtt{X1} = \mathtt{newvar}(), \ldots, \mathtt{Xn} = \mathtt{newvar}() \\ \mathtt{E} = \mathit{Compile}\ R\ \mathit{seq}\ \Gamma \\ \Gamma = [\alpha.p_1{:=}\mathtt{X1}, \ldots, \alpha.p_n{:=}\mathtt{Xn}] \\ D_{\mathcal{S}}[P, R, c] = \mathit{seq} \end{array}$$

- Pour chaque entrée dans la table $FC_{\mathcal{S}}$ telle que $FC_{\mathcal{S}}[P, R] = \mathtt{f}$, je crée la déclaration de fonction suivante:

  ```
  f X1 ... Xn = fun
  | c Y1 ... Ym -> E
  | ...
  ```

  avec les définitions suivantes:

  $$\begin{array}{l} \overline{P} = \{p_1, \ldots, p_n\} \\ \mathtt{X1} = \mathtt{newvar}(), \ldots, \mathtt{Xn} = \mathtt{newvar}() \\ \Gamma_0 = [\alpha.p_1{:=}\mathtt{X1}, \ldots, \alpha.p_n{:=}\mathtt{Xn}] \end{array}$$

  Il y a un cas pour chaque constructeur c tel que $c \in TS_{\mathcal{S}}[P, R]$, et on a pour chacun d'eux les définitions suivantes:

  $$\begin{array}{l} \mathtt{Y1} = \mathtt{newvar}(), \ldots, \mathtt{Ym} = \mathtt{newvar}() \\ \mathtt{E} = \mathit{Compile}\ R\ \mathit{seq}\ \Gamma \\ \Gamma = \Gamma_0[\alpha.1{:=}\mathtt{Y1}, \ldots, \alpha.m{:=}\mathtt{Ym}] \\ D_{\mathcal{S}}[P, R, c] = \mathit{seq} \end{array}$$

9.3.6 Exemple

Dans cette section, je présente un exemple complet, avec un programme équationnel qui contient les fonctions `map` et `length`, et une fonction `main` qui exécute `map` avec en paramètre `length`.

La fonction `length` correspond au programme équationnel suivant:

$$\mathcal{P}rg_1 = \left\{ \begin{array}{ccll} \mathit{length} & \to & \alpha.r = \alpha.L_1.l & (1) \\ " & \to & \alpha.L_1 = \alpha.p & (2) \\ \mathit{cons} & \to & \alpha.l = (+\ 1\ \alpha.2.l) & (3) \\ \mathit{nil} & \to & \alpha.l = 0 & (4) \end{array} \right\}$$

La fonction `map` correspond au programme équationnel suivant:

$$\mathcal{P}rg_2 = \left\{ \begin{array}{lcll} cons & \rightarrow & \alpha.m = (cons\ \alpha.L_2.r\ \alpha.2.m) & (5) \\ \text{''} & \rightarrow & \alpha.2.f = \alpha.f & (6) \\ \text{''} & \rightarrow & \alpha.L_2 = \alpha.f & (7) \\ \text{''} & \rightarrow & \alpha.L_2.p = \alpha.1 & (8) \\ nil & \rightarrow & \alpha.m = (nil) & (9) \end{array} \right\}$$

Le programme équationnel complet correspond à:

$$\mathcal{P}rg = \mathcal{P}rg_1 \cup \mathcal{P}rg_2 \cup \left\{ \begin{array}{lcll} main & \rightarrow & \alpha.r = \alpha.L_3.m & (10) \\ \text{''} & \rightarrow & \alpha.L_3 = \alpha.p & (11) \\ \text{''} & \rightarrow & \alpha.L_3.f = (length) & (12) \end{array} \right\}$$

Les stratégies calculées pour ce programme sont alors les suivantes (les numéros font référence aux équations ci-dessus):

- Pour la paire (p,r):

$$\left\{ \begin{array}{llll} (p,r, & length, & [(2);(\emptyset,l,\alpha.L_1);(1)]) \\ (p,r, & main, & [(11);(12);(f,m,\alpha.L_3);(10)]) \end{array} \right.$$

- Pour la paire $(\emptyset,l)$:

$$\left\{ \begin{array}{llll} (f,m, & cons, & [(\emptyset,l,\alpha.2);(3)]) \\ (f,m, & nil, & [(4)]) \end{array} \right.$$

- Pour la paire (f,m):

$$\left\{ \begin{array}{llll} (f,m, & cons, & [(6);(f,m,\alpha.2);(7);(8);(p,r,\alpha.L_2);(5)]) \\ (f,m, & nil, & [(9)]) \end{array} \right.$$

On a alors, suivant les définitions de la section précédente, les tables suivantes:

$$\begin{array}{l} F_{\mathcal{S}} = \{(p,r)\} \\ FP_{\mathcal{S}}[main] = (p,r,\mathtt{f0}) \\ FP_{\mathcal{S}}[length] = (p,r,\mathtt{f1}) \\ FC_{\mathcal{S}}[\emptyset,l] = \mathtt{f2} \\ FC_{\mathcal{S}}[f,m] = \mathtt{f3} \end{array}$$

On effectuant toutes les compilations nécessaires, on a le programme suivant:

```
let f0 x1 = (f3 f1 x1)
let f1 x1 = (f2 x1)
let f2 = fun
| cons y1 y2 ->
    let z1 = (f2 y2) in
    let z2 = (+ 1 z1) in
    z2
| nil ->
    let z1 = (0) in
    z1
let f3 x1 = fun
| cons y1 y2 ->
    let z1 = x1 in
    let z2 = (f3 z1 y2) in
    let z3 = x1 in
    let z4 = y1 in
    let z5 = (z3 z4) in
    let z6 = (cons z5 z2) in
    z6
| nil ->
    let z1 = (nil) in
    z1
```

On retirant les "`let`... `in`" qui sont ici inutiles (et donc éliminés par tous les compilateurs fonctionnels), on trouve le programme suivant:

```
let f0 x1 = (f3 f2 x1)

let f2 = fun
| cons y1 y2 -> (+ 1 (f2 y2))
| nil -> (0)

let f3 x1 = fun
| cons y1 y2 -> (cons (x1 y1) (f3 x1 y2))
| nil -> (nil)
```

9.4 Équivalence & Correction

Définir la correction de cette traduction n'est pas immédiat. Comparer deux programmes écrits dans des formalismes très différents pose toujours un problème. Tout d'abord, on peut vérifier un petit lemme qui nous sera bien commode: pour toute valeur fonctionnelle, et dans tout environnement de traduction Γ, on a:

$$[\![v]\!]\ x, \Gamma = \overline{v}$$

où $\overline{v}$ remplace juste les fermetures par leur constructeur équationnel correspondant:

$$\begin{array}{lcll}\overline{(\varphi\ v_1 \ldots v_n)} & = & (\varphi\ \overline{v_1} \ldots \overline{v_n}) & (\forall \varphi \in \mathcal{C}ons \cup \mathcal{P}rim) \\ \overline{(f\ v_1 \ldots v_k)} & = & (f_k\ \overline{v_1} \ldots \overline{v_k}) & \end{array}$$

Dans la sémantique d'un programme fonctionnel, je fais intervenir une relation $\rightarrow_\gamma$, où cette fois γ est un environnement d'exécution qui associe des valeurs à des variables fonctionnelles. Je définis alors une conversion entre les deux types d'environnement de manière suivante:

$$\begin{array}{ll}\overline{\gamma}(x)(y) & = \{y = \overline{\gamma(x)}\} \quad \text{si} \quad \gamma(x) \neq \bot \\ \overline{\gamma}(x)(y) & = \Gamma_D(x)(y) \quad \text{sinon}\end{array}$$

Si maintenant je note $\mathcal{P}rg$ la traduction du programma fonctionnel considéré, je peux alors prouver le théorème suivant:

Théorème 9.4.1 (Traduction d'un programme fonctionnel)
Pour toute expression fonctionnelle e et tout environnement γ, si on a:

$$e \rightarrow_\gamma v$$

alors on a pour toute variable équationnelle y:

$$[\![e]\!]\ y, \overline{\gamma}\ (\rightarrow_{\mathcal{P}rg}^{sat})^*\ \{y = \overline{v}\}$$

Cette implication utilise la notion de dérivation saturée définie au chapitre 8, mais il suffit de savoir que $(\rightarrow_{\mathcal{P}rg}^{sat})^*$ est un cas particulier de $\Rightarrow_{\mathcal{P}rg}$.

Une forme équivalente de ce théorème est l'énoncé suivant, qui convaincra peut-être mieux le lecteur que la traduction est "correcte". Pour toutes valeurs v_1, v_2 et v_3, on a:

$$(v_1, v_2)[\![\mathcal{P}rg]\!]^{sat}_{(arg,call)}(v_3) \quad \Leftrightarrow \quad (v_1\ v_2) \rightarrow_\varepsilon v_3$$

Les preuves de ces théorèmes ne sont pas difficiles et suivent simplement à la lettre les définitions de la traduction donnée à la section 9.2. Je ne m'étends donc pas plus sur le sujet.

Chapitre 10

Outils Algébriques

Ce chapitre concerne l'ensemble des notions mathématiques élémentaires qui sont utilisées dans cette thèse. Je profite de ce chapitre, nécessaire à mes lecteurs qui ne seraient pas familiers de la manipulation des ensembles et des relations, pour fixer quelques notations et un peu de vocabulaire. Ce chapitre contient également tous les algorithmes de base et une description de quelques problèmes d'implémentation.

A chaque fois qu'un chapitre utilisera une notion décrite dans celui-ci, j'en ferai explicitement la mention le moment venu dans celui-là. Le présent chapitre peut donc être consulté au fur et à mesure des besoins. Les différentes parties qui le composent sont plus ou moins indépendantes, tout en conservant une certaine linéarité.

10.1 Relations

Quelques notations au niveau des relations.

Somme: La somme de deux relations $\mathcal{R}_1$ et $\mathcal{R}_2$ est la relation définie par:

$$x\ (\mathcal{R}_1 + \mathcal{R}_2)\ y \Leftrightarrow x\ \mathcal{R}_1\ y \text{ ou } x\ \mathcal{R}_2\ y$$

Séquence: La mise en séquence de deux relations $\mathcal{R}_1$ et $\mathcal{R}_2$ est la relation définie par:

$$x\ (\mathcal{R}_1 ; \mathcal{R}_2)\ y \Leftrightarrow (\exists z)\ x\ \mathcal{R}_1\ z \text{ et } z\ \mathcal{R}_2\ y$$

Puissance: On définit la puissance k-ième d'une relation $\mathcal{R}$ par:

$$\begin{array}{lcl} \mathcal{R}^0 & = & \mathrm{Id} \\ \mathcal{R}^{k+1} & = & (\mathcal{R}^k ; \mathcal{R}) \end{array}$$

Fermeture: La fermeture transitive d'une relation $\mathcal{R}$ est définie par:

$$x \ \mathcal{R}^* \ y \Leftrightarrow (\exists m) \ x \ \mathcal{R}^m \ y$$

10.2 Points Fixes

Cette section fait un bref résumé de techniques classiques concernant le calcul des points fixes. Les résultats proposés sont tous issus d'une théorie bien connue: l'*interprétation abstraite*.

Je considère donc un ensemble $\mathcal{E}$ partiellement ordonné par $(\sqsubset)$. Je note $a \sqcap b$ le plus grand minorant de a et b, et $a \sqcup b$ leur plus petit majorant. Je suppose que $\mathcal{E}$ est déjà complété par les éléments $\perp$ et $\top$ tels que $\sqcap$ et $\sqcup$ soient toujours définis.

Soit f une fonction de $\mathcal{E}$ dans lui-même. Un point fixe de f est un élément x tel que $f(x) = x$. Le plus grand point fixe est noté gfix f. On dit que f est monotone quand $x \sqsubset y$ implique $f(x) \sqsubset f(y)$. Un théorème important est alors le suivant:

Théorème 10.2.1 *Si f est monotone alors la suite définie par:*

$$x_n = f^n(\top)$$

converge vers gfix f.

On peut calculer le plus grand point fixe qui soit plus petit qu'une certaine valeur x_0. Pour cela, il suffit de considérer la suite suivante:

$$x_{n+1} = x_n \sqcap f(x_n)$$

Si $\mathcal{E}$ est fini, alors cette suite converge en un nombre fini de pas, donc il existe m tel que $x_\infty = x_m$. Je note ce calcul (fix f x_0). On aura donc:

$$\text{gfix } f = \text{fix } f \ \top$$

En pratique, on utilise beaucoup la fonction fix qui est facile à implémenter. Notez au passage que, si $f(x_0) \sqsubset x_0$, alors on a $x_n = f^n(x_0)$, ce qui simplifie encore le calcul.

10.3 Relations Déductives

Je m'intéresse ici à des relations entre des ensembles. Soit donc $\mathcal{E}$ un ensemble dont on s'intéresse aux parties.

Définition: $\rightarrow$ est déductive si et seulement si elle est transitive et si, pour tout $A, B, C \subset \mathcal{E}$ on a:

$$\begin{array}{l} A \cup B \rightarrow A \\ A \rightarrow B \text{ et } A \rightarrow C \Rightarrow A \rightarrow B \cup C \end{array}$$

Une relation déductive vérifie beaucoup de propriétés de monotonie. En particulier, elle se manipule exactement comme l'implication logique, le connecteur *et* étant remplacé par l'union ensembliste. Ainsi:

Théorème: La relation $\rightarrow$ vérifie (entre autres...):

- Identité: $A \rightarrow A$.
- Confluence: si $A \rightarrow B$ et $A \rightarrow C$ alors $A \rightarrow B \cup C$.
- Transitivité: si $A \rightarrow B$ et $B \rightarrow C$ alors $A \rightarrow C$.
- Affaiblissement à droite: si $A \rightarrow B$ alors pour tout $C \subset B$, $A \rightarrow B$.
- Affaiblissement à gauche: si $A \rightarrow B$ alors pour tout $C \supset A$, $C \rightarrow B$.

Définition: $\rightarrow$ est sous déductive si elle est transitive et si, pour tout A, B, C inclus dans $\mathcal{E}$ on a:

$$A \rightarrow B \Rightarrow A \cup C \rightarrow B \cup C$$

Définition: $\rightarrow$ étant sous-déductive, son extension déductive est la relation $\rightarrow^{ded}$ définie par:

$$A \rightarrow^{ded} B \Leftrightarrow (\exists C \supset B) \quad A \rightarrow C$$

Théorème: une relation déductive est également sous déductive. L'extension déductive d'une relation sous déductive est bien entendu déductive.

Preuve: sans difficulté, par union et transitivité.

Théorème: les relations déductives sont stables par addition (+), composition (;) et donc par fermeture transitive (*). De même pour les relations sous déductives.

Preuve: sans difficulté, par cas et par récurrence.

10.4 Réécriture

10.4.1 Algèbre de termes

On a déjà vu cette notion au chapitre 2. Plus généralement, pour une algèbre de terme $\mathcal{A}lg_T$, on se donne un ensemble de constructeurs $\mathcal{C}ons_T$, et l'algèbre est le plus petit ensemble contenant tous les $(f\ t_1 \dots t_n)$ avec $f \in \mathcal{C}ons_T$ et pour tout i, $t_i \in \mathcal{A}lg_T$.

10.4.2 Égalité structurelle

Je définie la relation d'égalité structurelle (ou égalité forte) classique pour les algèbres de termes. Je la note $=_{struct}$, et elle est définie par:

$$(\forall s, t \in \mathcal{A}lg_T) \quad s =_{struct} t \Leftrightarrow (\exists f \in \mathcal{C}ons_T) \left\{ \begin{array}{l} s = (f\ s_1 \dots s_n) \\ t = (f\ t_1 \dots t_n) \\ (\forall i) \quad s_i =_{struct} t_i \end{array} \right.$$

10.4.3 Substitutions

Soit σ une substitution, je note $\overline{\sigma}$ son support, c'est à dire l'ensemble $\overline{\sigma} = \{a \mid \sigma(a) \neq \bot\}$.

Soient σ_1 et σ_2 deux substitutions. Si elles sont identiques sur leur support commun, c'est à dire $\sigma_1(x) = \sigma_2(x)$ pour tout $x \in \overline{\sigma_1} \cap \overline{\sigma_2}$), alors je définis $\sigma_1 \sqcup \sigma_2$ la substitution σ_3 définie par: $\sigma_3(x) = \sigma_1(x)$ si $x \in \overline{\sigma_1}$, et $\sigma_3(x) = \sigma_2(x)$ si $x \in \overline{\sigma_2}$. Dans le cas contraire, je note $\sigma_1 \sqcup \sigma_2 = \bot$.

Avec cette relation $\sqcup$, il est facile de définir une ordre sur les substitutions de la manière suivante:

$$\sigma_1 > sigma_2 \Leftrightarrow \sigma_1 \sqcup \sigma_2 = \sigma_1$$

10.4.4 Unification

Une unification des termes t_1 et t_2 consiste à trouver la plus petite substitution σ telle que $\sigma(t_1) = \sigma(t_2)$, si elle existe. On peut représenter cette opération en notant $\mathrm{unif}(t_1, t_2)$. Il existe de nombreux algorithmes permettant de calculer cette opération. Je ne les détaille pas ici.

10.4.5 Relation de réécriture

Une relation de réécriture une relation $\triangleright$ entre termes qui soit stable par substitution. On veut donc avoir pour toute substitution σ la relation :

$$(\forall t, t') \quad t \triangleright r' \Rightarrow \sigma(t) \triangleright \sigma(t')$$

Un cas de particulier de relation de relation de réécriture est intéressant. Soit en effet un ensemble de paires de termes : $R \subset \mathcal{T}erms * \mathcal{T}erms$. On peut définir une relation de réécriture ainsi :

$$t \triangleright t' \Leftrightarrow \left\{ \begin{array}{c} (\exists (t_0, t'_0) \in R) \\ \sigma = \mathrm{unif}(t_0, t) \\ t' = \sigma(t'_0) \end{array} \right.$$

Par exemple, si on a :

$$R = \{\ ((+\ a\ b), (+\ b\ a))\ \}$$

Alors $\triangleright$ représente la commutativité de $+$.

10.5 Parties Coniques

10.5.1 Définition et propriétés

Je m'intéresse ici à un ensemble $\mathcal{E}$ partiellement ordonné par une relation $<$.

Définition: le cône issu de $a \in \mathcal{E}$ est l'ensemble suivant:

$$a^{\uparrow} = \{b \in \mathcal{E} \mid b > a\}$$

Par extension, le cône issu d'une partie A est la réunion des cônes issus de tous ses éléments:

$$A^{\uparrow} = \{b \in \mathcal{E} \mid (\exists a \in A)\ b > a\}$$

Définition: une partie $A \subset \mathcal{E}$ est conique si et seulement si:

$$(\forall a \in A) \quad y > a \Rightarrow y \in A$$

Il est équivalent de dire qu'une partie A est conique si et seulement si $A^{\uparrow} = A$. Par ailleurs, l'ensemble des parties coniques est noté $\mathcal{E}^{\uparrow}$. On a quelques bonnes propriétés algébriques sur les coniques, par exemple:

$$\begin{array}{l} (A \cup B)^{\uparrow} = A^{\uparrow} \cup B^{\uparrow} \\ (A \cap B)^{\uparrow} \subset A^{\uparrow} \cap B^{\uparrow} \end{array}$$

En corollaire, on obtient les propriétés suivantes:

Théorème 10.5.1 *Si A et B sont coniques, alors $A \cup B$ et $A \cap B$ aussi.*

Preuve: sans difficulté en appliquant les définitions.

Le problème des parties coniques est qu'elles sont en général très grosse. Mon but est ici d'en chercher des représentations plus petites. J'introduis donc la notion de générateur.

Définition: une partie $G \subset \mathcal{E}$ est un générateur de A si et seulement si $G \uparrow = A$. Remarque: A est alors conique.

En fait, quand on a une partie conique A, on cherche un générateur B qui soit le plus petit possible. Cette notion est intéressante car elle permet de représenter des gros ensemble comme un cône issu d'un petit nombre d'éléments. Dans ma thèse, j'aurais plusieurs fois le cas d'ensembles stables par une certaine relation d'ordre (souvent partiel). Les propriétés qui suivent me permettront alors faire des calculs plus facilement sur de tels ensembles, en cherchant à les représenter comme des cônes issus du plus petit nombre d'éléments possible.

Définition: Une partie $L \subset \mathcal{E}$ est libre si et seulement si, pour tous $x, y \in L$, alors on a $x \vee y \neq \perp \Rightarrow x = y$.

Définition: Une partie libre et génératrice de A est une base de A.

Théorème 10.5.2 *Soit B un générateur de A. B est une base de A si et seulement si, pour tout générateur G de A, il existe une application φ de $\mathcal{E}$ dans lui-même telle que $\varphi(G) = B$.*
Un corollaire pour le cas fini est qu'une base est un générateur de cardinal minimal.

Preuve: Je montre l'équivalence en deux parties.

Sens direct: Soit B une base, soit G un générateur non libre. Donc il existe $a \in B$ et $b \in B$ tels que $a \vee b = c$. Il me suffit de montrer que nécessairement $a = b$. Je considère alors $G' = B - \{a, b\} \cup \{c\}$. Je laisse ici au lecteur le soin de démontrer que G' est aussi un générateur de A. Donc,comme B est une base, il existe φ telle que $\varphi(B) = G'$. Soit maintenant ψ définie par $\psi(a) = c$, $\psi(b) = c$ et $\phi(x) = x$ sinon. On a donc $\psi(G') = B$. Donc $\psi \circ \varphi$ et $\varphi \circ \psi$ sont bijectives. Donc ψ est injective, donc $a = b$.

Réciproque: Supposons B libre. Soit G un générateur de A. On a $\forall x \in G$, $x \in G \uparrow$, donc $x \in B \uparrow$ car B est générateur de A. Donc $\exists b \in B$ tel que $x > b$. On a ainsi une fonction φ de G dans B telle que, $\varphi(G) \subset B$ et $(\forall x)\ x > \varphi(x)$. Il ne reste plus qu'à montrer que φ est surjective. Soit donc $x \in B$. G est générateur donc.

Ce théorème définit ainsi une propriété caractéristique des bases, dont on se servira beaucoup pour calculer la base d'une partie conique.

Théorème: Si B est une base de A, alors B est unique.

Preuve: Soient $B_1 = \{a_i\}_{i \leq n}$ et $B_2 = \{b_i\}_{i \leq n}$ deux bases. B_1 est générateur de A et $B_2 \subset A$, donc $(\forall i)(\exists k)\ b_i > a_k$. Supposons alors que $b_i > a_k$ et $b_j > a_k$, alors $b_j \vee b_i > a_k$. Or B_2 est générateur, donc d'après le lemme, $i = j$.
Donc il existe une substitution σ telle que $(\forall i)\ b_i > a_{\sigma(i)}$. Et de même, il existe

une substitution σ' telle que $(\forall i)\ a_j > b_{\sigma'(j)}$.
Donc $b_i > b_{\sigma\circ\sigma'(i)}$. Donc $b_i \vee b_{\sigma\circ\sigma'(i)} = b_{\sigma\circ\sigma'(i)}$. Or B_2 est générateur, donc a nécessairement $\sigma \circ \sigma' = id$ grâce au lemme.
On a alors $a_{\sigma(i)} > b_{\sigma'(\sigma(i))} = b_i$.
Finalement, $b_i > a_{\sigma(i)}$ et $b_i < a_{\sigma(i)}$, donc $b_i = a_{\sigma(i)}$, donc $B_1 = B_2$.

10.5.2 Calcul des bases

Dans cette section, je présente un algorithme qui permet de calculer la base d'une partie conique. Plus précisément, cet algorithme calcule le générateur d'un ensemble $A^{\uparrow}$, où A est une partie finie.

La clef de l'algorithme réside dans une fonction d'addition, *addconify*. Si G est un générateur d'un ensemble de la forme $B^{\uparrow}$, alors *addconify* x G est un générateur de la partie conique $(B \cup \{x\})^{\uparrow}$.

Dès lors, si cette fonction est connue, on peut calculer le générateur de toute partie de la forme $A = \{a_1 \ldots a_n\}^{\uparrow}$. En effet, on a alors:

$$gen(A) = (addconify\ a_1\ addconify\ a_2\ (\ldots\ \emptyset))))$$

Voici donc ces deux algorithmes:

gen A	si $A = \emptyset$, alors $\emptyset$, sinon, avec $x \in A$ renvoyer *addconify* x $(gen(A - \{x\}))$.
addconify x G	si $G = \emptyset$, renvoyer $\{x\}$ sinon, avec $y \in G$: si $y \vee x = z$ renvoyer *addconify* z $(G - \{y\})$, sinon renvoyer $\{y\} \cup (addconify\ x\ (G - \{y\}))$

10.5.3 Union et Intersection des coniques

Faire des réunions de parties coniques dont on connaît les générateurs est aisée. En effet, on a:

$$gen\ (A \cup B) = gen\ ((gen\ A) \cup (gen\ B))$$

Pour les intersections, le problème est similaire mais pas vraiment dual. Ainsi, on a besoin de l'opérateur $\wedge$ qui est défini comme suit:

$$a \wedge b = gen\ (a^{\uparrow} \cap b^{\uparrow})$$

En général, $a \wedge b$ sera réduit à un élément ou bien l'ensemble vide. Mais il se peut que l'intersection de deux cônes ne soit pas réduit à un seul cône. On a donc:

$$gen\ (A \cap B) = gen\ \bigcup \{a \wedge b \mid a \in gen\ A, b \in gen\ B\}$$

Cette manière de calculer les unions et les intersections de parties coniques me sera très utile pour certains algorithmes. En pratique, ces algorithmes de calcul sont tous polynômiaux en la taille des générateurs de A et B. Le degré exact du polynôme varie en fonction de l'implémentation des ensembles que l'on choisit. Mais il faut garder à l'esprit que, en pratique, les générateurs n'auront jamais beaucoup d'éléments, alors que les parties coniques à considérer sont souvent de taille astronomique. Donc tout ces calculs se font très vite par rapport au résultat escompté.

10.6 Parties Compatibles Maximales

Le problème à résoudre ici est relativement simple. On se donne une relation symétrique $\mathcal{R}$ sur un ensemble $\mathcal{E}$, et on cherche les plus grandes parties de $\mathcal{E}$ qui ne contiennent que des éléments en relation par $\mathcal{R}$. Autrement dit, on cherche un ensemble $\mathcal{A}$ de parties de $\mathcal{E}$ tel que:

$$\begin{array}{l}(\forall A \in \mathcal{A})(\forall x, y \in A) \quad x\mathcal{R}y \\ (\forall A, B \in \mathcal{A}) \quad A \subset B \Rightarrow A = B\end{array}$$

Le but du jeu consiste bien sûr à chercher $\mathcal{A}$ le plus rapidement possible. La solution que j'utilise m'a été fournie par un collègue et elle est étonnement efficace. En voici l'algorithme, dénommé *maxcompat*.

maxcompat $\mathcal{R}$ $\mathcal{E}$	calcule les parties maximales.
1	calculer I, l'ensemble des (x, y) incompatibles par $\mathcal{R}$.
2	accumuler les parties calculées par *scan* I $\mathcal{E}$, en ne retenant que les parties maximales.
scan I A	si $I = \emptyset$, accumuler A.
	sinon, soient $(x, y) \in I$ et $I' = I - (x, y)$:
	si $x \notin A$ et $y \notin A$ alors accumuler *scan* I' A.
	sinon accumuler *scan* I' $(A - x)$ et *scan* I' $(A - y)$

Cet algorithme fonctionne, car il retire a chaque itération une des composantes qui rendrait une partie non compatible et laisse l'autre. En partant de la totalité de l'ensemble, on génère ainsi toute les séquences de plus grande taille. Mais on génère aussi des séquences plus petites, donc c'est pourquoi il faut les filtrer en les accumulant.

Notons d'ailleurs que lors de cette accumulation on ne retient que les plus grandes parties, c'est donc un cas particulier du calcul d'un générateur de parties coniques avec l'ordre $\supset$. En effet, si A est compatible alors $B \subset A$ est aussi compatible. Donc, si G contient les parties accumulées à un moment donné, et si on doit accumuler la partie A, il suffit d'utiliser la fonction (*addconify* A G) avec pour fonction $\vee$ la fonction (symétrique) définie par:

$$A \subset B \Leftrightarrow A \vee B = B$$

et qui vaut $\perp$ sinon. Cet algorithme fonctionne d'autant mieux que le nombre de paires incompatibles est faible, et il est dans tous les cas bien meilleur qu'un algorithme qui parcourrait l'ensemble des parties de $\mathcal{E}$.

10.7 Graphes

Un graphe est simplement une relation. Soit G un graphe, si on a $x\ G\ y$, je le noterai aussi en disant que l'arc $x \to y$ est dans G, soit encore $x \to y \in G$. De nombreuses opérations existent sur les graphes, et qui correspondent exactement aux opérations sur les relations. Je ne détaille pas ce point.

Je veux ici m'attarder sur une petite notation pour des graphes particulier. Soit un ensemble A et une relation G entre *des parties* de A et des *éléments* de A. De même que pour les graphes, si $X \subset A$ et $a \in A$, alors je noterai $X \to a \in G$ si on a la relation $X\ G\ a$.

Ce qui m'interesse ici, c'est que l'on peut voir G comme un graphe un peu spécial, ou plus exactement, on peut définir une notion de chemin entre les éléments de A. Soit en effet le graphe $\overline{G}$ défini par:

$$\overline{G} = \{x \to a \mid \exists (X \to a) \in G, \quad x \in X\}$$

Alors, je dirais que il existe un chemin entre a et b dans le "graphe" G si il existe un chemin entre a et b dans le graphe $\overline{G}$, c'est à dire si $a \to b \in (\overline{G})^*$. Plus simplement, je dirais que la fermeture de G est par extension le graphe $(\overline{G})^*$.

Chapitre 11

Conclusion

Dans cette thèse, j'ai essayé de répondre de manière unifiée à des problèmes connus de transformations de programmes dans différents formalismes. Je ne reviendrais pas sur ce point (Cf. Introduction). Même si je n'ai pas voulu me limiter à certaines transformations, je me suis quand même surtout intéressé aux plus difficiles et aux moins bien résolues dans le cas le plus général: l'élimination ou plutôt la simplification des compositions de fonctions.

Le problème de la transformation des compositions dans les programmes fonctionnels est un problème épineux pour de nombreuses raisons. Premièrement, la complexité et la diversité des situations rencontrées fait que certains algorithmes ne peuvent que couvrir une partie du problème, comme en témoignent les nombreuses transformations associées. Deuxièmement, ces compositions sont fréquentes dans la pratique. C'est sur ce dernier point que je voudrais m'attarder.

Généricité

J'avais fait connaissance avec les compositions de fonctions dans le cadre d'un premier travail qui portait sur la généricité. Le but était de définir des algorithmes, dits génériques, sur un type de structure le plus général possible. Ensuite, l'idée est qu'il "suffit" de composer ces algorithmes génériques avec des morphismes de types pour automatiquement en bénéficier sur d'autres types. Par exemple:

On veut itérer une opération sur des listes d'objets. Il y a malheureusement deux façons de faire cette itération, de gauche à droite ou bien l'inverse. Si on veut ces deux itérateurs sur les listes et sur les arbres, il faudrait écrire quatre fonctions:

- `fold_left_list`: itère une fonction sur une liste de la gauche vers la droite.
- `fold_right_list`: itère une fonction sur une liste de la droite vers la gauche.

- `fold_left_tree`: itère une fonction sur les feuilles d'un arbre de la gauche vers la droite.
- `fold_right_tree`: itère une fonction sur les feuilles d'un arbre de la droite vers la gauche.

Avec une technique générique, seule une fonction d'itération est nécessaire (en imaginant que les fonctions `rev` et `flat` soient déjà connues). En effet, écrivons un itérateur générique:

```
let fold f r = fun
| cons a b -> (fold (f r a) f b)
| nil -> r
```

Pour définir les quatre fonctions voulues, il suffit d'écrire:

```
let fold_left_list  f r x = fold f r x
let fold_right_list f r x = fold f r (rev nil x)
let fold_left_tree  f r x = fold_left_list f r (flat nil x)
let fold_right_tree f r x = fold_right_list f r (flat nil x)
```

Bien entendu, on y gagnerai beaucoup plus si la fonction `fold` était très compliquée. Mais il est tout de même surprenant qu'avec un si petit exemple, aucune méthode de déforestation classique ne parvient à simplifier les compositions proposées ici.

Avec la sémantique équationnelle, bien entendu, ces compositions sont facilement simplifiées, et on obtient bien les quatre itérateurs voulus dans leur forme la plus efficace en temps et en mémoire.

Pour des exemples plus complexes, on peut imaginer de nombreuses applications de cette programmation générique.

De la difficulté des compositions

Je voudrais terminer par une petite classification des problèmes de compositions, et en donner leur difficulté. Je distingue trois types de fonctions. Le premier, noté S (comme Simple), est l'ensemble des fonctions qui n'ont pas de paramètres (par exemple: `length`). Le second type, noté A, est celui des fonctions qui ont au moins un paramètre accumulatif, c'est à dire que le résultat de la fonction est construit sur ce paramètre (par exemple: `rev`, `flat`). Enfin, le dernier type regroupe les autres fonctions avec paramètres (par exemple: `map`).

Maintenant, je noterai par exemple (S.P) la composition d'une fonction du type P avec une fonction du type S, la fonction de type S étant calculée sur le résultat de la fonction de type P. On peut alors distinguer les catégories suivantes:

Cas faciles: les compositions (S.P), (P.S), (S.S), et (A.S) sont faciles à simplifier. On obtient directement les résultats par application d'une forme quelconque d'évaluation partielle, fonctionnelle ou non.

Cas difficiles: les compositions (S.A) sont difficiles. Elles correspondent au premier cas de précalculation décrit à la section 5.5, mais on peut tout de même effectuer cette transformation en restant dans le cadre fonctionnel (voir fin de la section 4.1).

Cas très difficiles: les compositions (P.A), (A.P), (P.P) et enfin (A.A) sont toutes très difficiles, impossibles à réaliser en restant dans un cadre purement fonctionnel, et correspondent au deuxième cas de précalculation décrit à la section 5.5.

Mais, la plupart des fonctions efficaces sont du type P ou A. Donc en général, on peut s'attendre à tomber dans le cas très difficile. Idéalement, on pourrait procéder en deux étapes: d'abord simplifier les compositions faciles ou difficiles par des méthode purement fonctionnelles, et ensuite seulement, utiliser des techniques équationnelles.

Travaux futurs, extensions

Dans le cadre de cette thèse, je n'ai pas voulu aborder le cas des constructions impératives. C'est un problème très délicat, et les analyses de programmes fonctionnels avec constructions impératives sont réputées difficile. Bien entendu, il existe des moyens canoniques de transformer un programme impératif en un programme purement fonctionnel: on peut par exemple modéliser l'état de la mémoire par un paramètre additionnel pour toutes les fonctions du programme, et en définissant des primitives d'affectation et d'allocation mémoire sur ce paramètre. On risque alors de ne pas simplifier certaines compositions. J'ai commencé à voir comment traiter ce type de cas, mais il reste encore de grandes zones d'ombres.

Chapitre 12

Index

Définitions et Théorèmes:

Bibliographie

[1] BOYLAND, J. Conditional attribute grammars. *ACM Trans. Program. Lang. Syst. 18*, 1 (Jan. 1996), 73–108.

[2] CHIN, W.-N., AND KHOO, S.-C. Better consumers for deforestation. In *Programming Languages: Implementations, Logic and Programs (PLILP '95)* (1995), S. D. Swierstra, Ed., vol. 982 of *LNCS*, Springer-Verlag, pp. 223–240.

[3] CORRENSON, L., DURIS, E., PARIGOT, D., AND ROUSSEL, G. Declarative program transformation:
a deforestation case-study. accepted at PPDP'99 and early version, 1999.

[4] DERANSART, P., JOURDAN, M., AND LORHO, B. *Attribute Grammars: Definitions, Systems and Bibliography*, vol. 323 of *Lect. Notes in Comp. Sci.* Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[5] DURIS, E. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle.* PhD thesis, Université d'Orléans, 1998.

[6] FARROW, R., MARLOWE, T. J., AND YELLIN, D. M. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages* (Albuquerque, NM, Jan. 1992), ACM press, pp. 223–234.

[7] GANZINGER, H., AND GIEGERICH, R. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction* (Montréal, June 1984), ACM press, pp. 157–170. Published as *ACM SIGPLAN Notices*, 19(6).

[8] GIEGERICH, R. Composition and evaluation of attribute coupled grammars. *Acta Informatica 25* (1988), 355–423.

[9] GILL, A. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Department of Computing Science, Glasgow University, Jan. 1996.

[10] GILL, A., LAUNCHBURY, J., AND JONES, S. L. P. A short cut to deforestation. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)* (Copenhagen, Denmark, June 1993), ACM Press, pp. 223–232.

[11] HAMILTON, G. W. Higher order deforestation. In *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)* (Aachen, Sept. 1996), H. Kuchen and S. D. Swierstra, Eds., vol. 1140 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 122–136.

[12] Hu, Z., Iwasaki, H., and Takeishi, M. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundations of Computer Science (MFCS'96)* (Cracow, Sept. 1996), vol. 1113 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 407–418.

[13] Hu, Z., Iwasaki, H., and Takeishi, M. Deriving structural hylomorphisms from recursive definitions. In *Proc.of the International Conference on Functional Programming (ICFP'96)* (Philadelphia, May 1996), ACM Press, pp. 73–82.

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

[15] Jeuring, J., and Jansson, P. Polytypic programming. In *Advanced Functional Programming* (1996), J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 68–114.

[16] Jourdan, M., and Parigot, D. Internals and Externals of the FNC-2 Attribute Grammar System. In *Attribute Evaluation Methods* (New York–Heidelberg–Berlin, June 1991), H. Alblas and B. Melichar, Eds., vol. 545 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 485–504. Prague.

[17] Kühnemann, A. Benefits of tree transducers for optimizing functional programs. In *Foundations of Software Technology & Theoretical Computer Science - FST & TCS'98, 18th Conference, Chennai, India, Proceedings* (December 1998), V. Arvind and R. Ramanujam, Eds., vol. 1530 of *LNCS*, Springer-Verlag, pp. 146–157.

[18] Kühnemann, A. Comparison of deforestation techniques for functional programs and for tree transducers. In *4th Fuji International Symposium on Functional and Logic Programming - FLOPS'99, Tsukuba, Japan, Proceedings* (November 1999), A. Middeldorp and T. Sato, Eds., vol. 1722 of *LNCS*, Springer-Verlag, pp. 114–130.

[19] Launchbury, J., and Sheard, T. Warm fusion: Deriving build-cata's from recursive definitions. In *Conf. on Func. Prog. Languages and Computer Architecture* (La Jolla, CA, USA, 1995), ACM Press, pp. 314–323.

[20] Malcolm, G. *Algebraic Types and Program Transformation.* PhD thesis, University of Gronigen, 1990.

[21] Meertens, L. Paramorphisms. *Formal Aspects of Computing 4*, 5 (1992), 413–424.

[22] Meijer, E., Fokkinga, M. M., and Paterson, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)* (Cambridge, Sept. 1991), vol. 523 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 124–144.

[23] Meijer, E., and Hutton, G. Bananas in space: Extending fold and unfold to exponential types. In *FPCA'95* (1995).

[24] Onoue, Y., Hu, Z., Iwasaki, H., and Takeichi, M. A calculational fusion system HYLO. In *Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi* (Le Bischenberg, France, Feb. 1997).

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

[26] Parigot, D. *Transformations, Évaluation Incrémentale et Optimisations des Grammaires Attribués: Le Système FNC-2.* PhD thesis, Université de Paris-Sud, Orsay, 1988.

[27] Parigot, D., and Jourdan, M. A complete bibliography on attribute grammars. `http://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html` Updated regularly. Contains around 1000 references to papers on Attribute Grammars. INRIA, France.

[28] Parigot, D., Roussel, G., Jourdan, M., and Duris, E. Dynamic Attribute Grammars. Rapport de recherche 2881, INRIA, May 1996.

[29] Parigot, D., Roussel, G., Jourdan, M., and Duris, E. Dynamic Attribute Grammars. In *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)* (Aachen, Sept. 1996), H. Kuchen and S. D. Swierstra, Eds., vol. 1140 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 122–136.

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

[31] Roussel, G., Parigot, D., and Jourdan, M. Coupling Evaluators for Attribute Coupled Grammars. In *5th Int. Conf. on Compiler Construction (CC' 94)* (Edinburgh, Apr. 1994), P. A. Fritzson, Ed., vol. 786 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 52–67.

[32] Seidl, H. Integer constraints to stop deforestation. In *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming* (Linköping, Sweden, Apr. 1996), H. R. Nielson, Ed., vol. 1058 of *Lecture Notes in Computer Science*, Springer, pp. 326–340.

[33] Seidl, H., and Sørensen, M. H. Constraints to stop deforestation. *Science of Computer Programming* (1997). to appear.

[34] Seidl, H., and Sørensen, M. H. Constraints to stop higher-order deforestation. In *24th ACM Symp. on Principles of Programming Languages* (1997), pp. 400–413.

[35] Sheard, T. A type-directed, on-line partial evaluator for a polymorphic language. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97* (1997), ACM press.

[36] Sheard, T., and Fegaras, L. A fold for all seasons. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)* (Copenhagen, Denmark, June 1993), ACM Press, pp. 233–242.

[37] Sørensen, M. H. A grammar-based data-flow analysis to stop deforestation. In *Colloquium on Trees in Algebra and Programming* (1994), S. Ti-

son, Ed., vol. 787 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 335–351.

[38] SØRENSEN, M. H., GLÜCK, R., AND JONES, N. D. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *European Symposium on Programming* (1994), D. Sannella, Ed., vol. 788 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 485–500.

[39] SWIERSTRA, S. D., AND VOGT, H. H. Higher order attribute grammars: a merge between functional and object oriented programming. report RUU–CS–90–12, Utrecht University, Mar. 1990.

[40] TAKANO, A., AND MEIJER, E. Shortcut deforestation in calculational form. In *Conf. on Func. Prog. Languages and Computer Architecture* (La Jolla, CA, USA, 1995), ACM Press, pp. 306–313.

[41] VOGT, H. H., SWIERSTRA, S. D., AND KUIPER, M. F. Higher order attribute grammars. In *ACM SIGPLAN '89 Conf. on Progr. Lang. Design and Implementation* (Portland, OR, July 1989), ACM press, pp. 131–145. Published as *ACM SIGPLAN Notices*, 24(7).

[42] VOGT, H. H., SWIERSTRA, S. D., AND KUIPER, M. F. On the efficient incremental evaluation of higher order attribute grammars. report RUU–CS–90–36, Utrecht University, Dec. 1990.

[43] WADLER, P. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming (ESOP '88)* (Nancy, Mar. 1988), H. Ganzinger, Ed., vol. 300 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pp. 344–358.

[44] WADLER, P. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science* (1990), vol. 73, pp. 231–248. (Special issue of selected papers from 2'nd European Symposium on Programming).