Interprétations polynomiales et applications aux λ-calculs avec substitutions explicites

Xavier Urbain

22 novembre 1996

Admháil...

Puisque ce stage est parvenu à son terme, du moins dans la durée, je voudrais tout d'abord remercier Claude Marché qui a eu la gentillesse de le rendre possible et la patience de me supporter durant ces quatre semaines. J'aimerais ensuite remercier la dynamique équipe des DÉMONS qui m'a accueilli avec beaucoup de sympathie. Je souhaiterais enfin adresser, en forme de clin d'œil, ma gratitude à tous ceux qui m'ont permis de goûter aux délices du restaurant du CESFO.

J'ai énormément appris, mais que ce mois était court !

Table des matières

Chapitre 1

En guise d'introduction

S'il est toujours employé, et avec succès, en mathématique et en informatique, le remplacement des termes par des termes équivalents dans une équation constitue un schéma de résolution fortement non-déterministe. On peut lever une première source de non-déterminisme en orientant les égalité, c'est-à-dire en ordonnant le remplacement des termes. Une deuxième source peut encore être levée si on bénéficie d'un système où chaque élément peut se réécrire en un unique terme qui n'est qu'un « membre gauche », ce qu'on appelle une forme normale. Devant l'importance que revêt le problème de l'existence d'une démarche permettant de résoudre les équations de manière efficace, la réécriture (voir [10] pour une introduction complète) apporte une forme de solution, formalisée à souhait. Voyons, pour la définir, quelques propriétés sur les relations.

Définition 1 *Une relation $\longrightarrow$ sur un ensemble $\mathcal{E}$ est* ***noétherienne*** *s'il n'existe pas de chaîne infinie d'éléments en relation :*

$$\nexists (x_n)_{n\in\mathbb{N}} : x_0 \longrightarrow x_1 \longrightarrow x_2 \longrightarrow \dots$$

Définition 2 *Une relation $\longrightarrow$ est dite* ***confluente*** *si :*

$$s \xleftarrow{\star} t \xrightarrow{\star} u \Rightarrow \exists v : s \xrightarrow{\star} v \xleftarrow{\star} u.$$

On peut maintenant définir la notion de réécriture :

Définition 3 *Une* ***règle*** *est un couple de termes (s,t) noté $s \longrightarrow t$.*

Soit R un ensemble de règles, on note $s[t]_p$ le remplacement de s par t à la position p, les substitution sont postfixes.

Définition 4 *La* ***relation de réécriture*** *par R, notée $\underset{R}{\longrightarrow}$, est la plus petite relation terme-compatible qui contienne R :*

$$s \underset{R}{\longrightarrow} t \Leftrightarrow \exists \sigma, u, p \left\{ \begin{array}{l} s = u[l\sigma]_p \\ t = u[r\sigma]_p \end{array} \right. \quad \text{où } l = r \in R.$$

Le problème est que, pour l'instant, rien ne s'oppose à ce qu'on applique à l'infini des règles de réécriture, et dans ce cas, on ne peut pas faire grand'chose. La notion clef est maintenant celle de terminaison, c'est elle qui va garantir l'arrêt des applications.

Définition 5 *On dira qu'un système de règles R* ***termine*** *ou* ***est à terminaison finie*** *si la relation $\xrightarrow[R]{}$ est noétherienne.*

La terminaison est indécidable.

Il faut maintenant prouver qu'un système termine. En appelant **ordre de réécriture** un ordre qui vérifie relation de la définition 4, on peut définir un **ordre de réduction** sur un ensemble de termes $\mathcal{T}$ comme un ordre de réécriture bien fondé sur $\mathcal{T}$. On a alors qu'un système R termine si chacune des règles de R est contenue dans un ordre de réduction (1).

Il existe quantité de méthodes pour prouver la terminaison. Un survol de ces techniques peut être trouvé dans [9]. La démarche couramment usitée consiste à affecter à chaque terme une valeur de complexité. Si cette valeur (positive) décroît au moins de 1 à chaque pas de réécriture, alors la terminaison est assurée. On peut dans certains cas considérer la « longueur » (le nombre de symboles) des membres droit et gauche, le nombre d'apparitions d'un symbole donné ou encore associer à chaque terme un entier. En 1979, Lankford se fait le porte drapeau de l'interprétation des termes par des fonctions polynomiales [11] permettant ainsi de prouver la terminaison de systèmes tels que (Associativité + Endomorphisme). Justifions cette méthode :

Soit $T(\mathcal{F}, x_1, \ldots, x_n)$ l'ensemble des termes sur $x_1, \ldots, x_n$. On note E l'ensemble des fonctions de $\mathbb{N}^n \to \mathbb{N}$. On suppose définie pour chaque fonction k-aire $f \in \mathcal{F}_k \subseteq \mathcal{F}$ une interprétation $[\![f]\!]$ en polynôme à k indéterminées. On peut alors définir sur E une $\mathcal{F}$ algèbre $\mathbb{N}_n$ de la manière suivante : si $p_1, \ldots, p_k \in E$ alors
$f_{\mathbb{N}_n}(p_1, \ldots, p_k)(x_1, \ldots, x_n) = [\![f]\!](p_1(x_1, \ldots, x_n), \ldots, p_k(x_1, \ldots, x_n))$. En posant $[\![x_i]\!] = x_i$, on a une extension unique d'un morphisme de $T(\mathcal{F})$ vers $\mathbb{N}_n$ imposée sur E. On la note de même $[\![\cdot]\!]$.

Il existe sur E un ordre partiel défini par $s < t$ si et seulement si $\forall a_i \in \mathbb{N},\ s((a_1, \ldots, a_n) < t(a_1, \ldots, a_n)$. Cet ordre est bien fondé. On peut maintenant se servir du critère de terminaison (1).

Notre travail va consister en l'implantation d'un tel ordre dans le système C*i*ME. Après avoir décrit plus en détail les ordres polynomiaux, nous exposerons les grandes lignes de l'implantation. Nous donnerons enfin une application de cette méthode en vérifiant la terminaison de quelques λ-calculs avec substitutions explicites.

Chapitre 2

Interprétations polynomiales

Pour prouver la terminaison d'un système de réécriture, une méthode possible consiste à trouver une interprétation judicieuse des deux membres de chaque règle puis montrer que l'ordre sur les interprétations vérifie bien les propriétés désirées. Le problème du choix des interprétations est très difficile et ne sera pas abordé dans ce chapitre où, après avoir constaté l'impossibilité de l'existence d'une méthode générale, nous exposerons deux procédés qui peuvent, dans certains cas, donner une réponse à la question de l'ordre sur les polynômes.

2.1 Aperçu du problème

Étant donné un système $\{a_1 \to b_1, ..., a_n \to b_n\}$ on va devoir ordonner les interprétations de sorte que, pour l'ordre $\succ$ sur les interprétations, on ait $[\![a_i]\!] \succ [\![b_i]\!]$. L'ordre utilisé sur les polynômes est défini comme suit : un polynôme P est supérieur (pour $\succ$) à un polynôme Q si, pour tout point x à coordonnées entières ≥ 2, on a $P(x) \geq Q(x)$. Le principal problème est qu'il n'existe pas d'algorithme qui permet de déterminer le plus grand de deux polynômes, cela est une conséquence de l'indécidabilité du dixième problème de Hilbert.

2.1.1 Hilbert et les ordres polynomiaux

Nous allons maintenant énoncer les idées qui mènent à l'impossibilité de trouver une méthode générale donnant en un temps fini une réponse au problème de l'ordonnancement de deux polynômes.

Le dixième problème de Hilbert. C'est lors de sa conférence au Congrès international des mathématiciens, en 1900, que Hilbert a posé la question de l'existence d'un « procédé régulier » qui, pour toute équation diophantienne donnée explicitement, permet par un nombre fini d'opérations de décider si, oui ou non, cette équation admet une solution entière. Hilbert ne définissant pas rigoureusement ce qu'il appelle un « procédé régulier », il a fallu attendre les années quarante pour qu'émerge une formulation précise de ce problème.

Une démonstration de la réponse (négative) à ce problème a été trouvée en 1970 par Matijasevič en terminant la preuve d'un théorème fondamental énoncé par Julia Robinson en 1952.

Démonstration de l'indécidabilité. Il suffit de regarder les solutions $(x_1, \ldots, x_n) \in \mathbb{N}^n$. En effet, si on avait une méthode pour des solutions dans $\mathbb{Z}^n$, on pourrait en déduire une méthode sur $\mathbb{N}^n$ grâce au théorème de Lagrange dit « des quatre carrés », en cherchant les solutions de $P(u_1^2 + v_1^2 + w_1^2 + t_1^2, \ldots, u_n^2 + v_n^2 + w_n^2 + t_n^2)$.
Tout d'abord quelques définitions et résultats...

Définition 6 *On appelle* ***équation diophantienne*** *une équation de la forme* $f(x_1, \ldots, x_n) = 0$ *où* f *est un polynôme à coefficients entiers.*

Définition 7 *Une relation* $\mathcal{R}$*,* k*-aire sur* $\mathbb{N}$*, est dite* ***diophantienne*** *si pour un polynôme* P *à* $k + n$ *indéterminées et à coefficients entiers on a :*

$$\mathcal{R} = \{(x_1, \ldots, x_k)/\exists y_1, \ldots, y_n \in \mathbb{N} : P(x_1, \ldots, x_k, y_1, \ldots, y_n) = 0\}.$$

Théorème 1 $\mathcal{R}$ *diophantienne est récursivement énumérable.*

En effet : soient $\mathcal{R}$ diophantienne et P le polynôme de la définition 2, soit $g_{\mathcal{R}}$ une application $\mathbb{N}^k \to \mathbb{N}$ définie par : $g_{\mathcal{R}}(x_1, \ldots, x_k) = 0 \; \forall (x_1, \ldots, x_k) \in \mathcal{R}$ et $g_{\mathcal{R}}(x_1, \ldots, x_k)$ indéfini sinon. Alors *il existe un algorithme qui calcule* $g_{\mathcal{R}}$. Cet algorithme consiste à ranger tous les $(y_{i_1}, \ldots, y_{i_n})$ en une suite et à calculer successivement $P(x_1, \ldots, x_k, y_{i_1}, \ldots, y_{i_n})$ jusqu'à ce qu'on obtienne 0 (si on l'obtient jamais). On a que $g_{\mathcal{R}}$ est Turing-calculable donc $\mathcal{R}$ est récursivement énumérable. cqfd.
le résultat suivant est le théorème fondamental, clef de la démonstration.

Théorème 2 (Robinson-Matijasevič) *Si* $\mathcal{R}$ *est récursivement énumérable alors elle est diophantienne.*

Une démonstration détaillée est disponible dans [5]. Nous nous bornerons à en exposer les principales étapes.
Remarquons tout d'abord que :

$$\left.\begin{array}{l} P_1 = 0 \\ P_2 = 0 \end{array}\right\} \Leftrightarrow P_1^2 + P_2^2 = 0 \tag{2.1}$$

$$P_1 = 0 \text{ ou } P_2 = 0 \Leftrightarrow P_1 \cdot P_2 = 0. \tag{2.2}$$

(2.1) et (2.2) entraînent que la classe des relations diophantiennes est close par union et intersection. On peut donc, en combinant des « définitions » diophantiennes avec « et », « ou » et des quantificateurs, en obtenir de nouvelles. On en tire :

- $x < y$ diophantienne car $\Leftrightarrow \exists z/(x + z + 1 = y)$
- $x \leq y$ diophantienne car $\Leftrightarrow \exists z/(x + z = y)$
- $x \equiv y \pmod z$ diophantienne car $\Leftrightarrow \exists t/(x + tz = y \vee y + tz = x)$

Lemme 1 *$z = x^y$ est diophantienne.*

Nous admettrons ce lemme, pour une démonstration détaillée voir [5].

Lemme 2 *$m = \binom{n}{k}$ et $m = n!$ sont diophantiennes.*

Pour démontrer la première affirmation il nous faut introduire un sous-lemme : *soit $\Phi(k, u, w)$ le coefficient de u^k dans le développement u-aire de w, alors $q = \Phi(k, u, w)$ est diophantienne.* Ce sous-lemme se prouve en constatant que la relation est définie par la vérification simultanée de :

$$\begin{cases} w = p + qu^k + r \\ q < u \\ p < u^k \\ r \equiv 0 \pmod{u^{k+1}} \end{cases}$$

En prenant $u > 2^n$ et en écrivant la formule du binôme pour $(u+1)^n$, on a que $\binom{n}{k}$ est le coefficient de u^k dans le développement de $(u+1)^n$, donc :

$$m = \binom{n}{k} \Leftrightarrow m = \Phi(k, 2^n + 1, (u+1)^n)$$

et la relation est diophantienne.
On admettra la seconde assertion.

Lemme 3 (théorème du quantificateur borné) *soit $P(u, x, y, h, z_1, \ldots, z_n)$ un polynôme, alors il existe un polynôme $Q(u, x, y)$ tel que :*

$$\begin{gathered} \forall k \leq x, \exists z_1, \ldots, z_n \leq y : P(u, x, y, k, z_1, \ldots, z_n) = 0 \\ \Updownarrow \\ \exists b_1, \ldots, b_n / \binom{b_1}{y+1} \equiv \ldots \equiv \binom{b_1}{y+1} \\ \equiv P(u, x, y, q! - 1, b_1, \ldots, b_n) \equiv 0 \left[\bmod \binom{q! - 1}{x+1} \right] \end{gathered}$$

où q représente la valeur de Q prise en (u, x, y), u pouvant être un vecteur.

Ce théorème est tiré d'un résultat de Davis, Putnam et Robinson. On pourra consulter [6].

Théorème 3 *Soit $\mathcal{R}$ une relation diophantienne et soit $\mathcal{S}$ une relation définie par : $\mathcal{S} = \{(x_1, \ldots, x_k, x) : \forall t \leq x, (x_1, \ldots, x_k, t) \in \mathcal{R}\}$, alors $\mathcal{S}$ est diophantienne.*

Démonstration : on a, pour le polynôme P de la définition,

$$\begin{aligned}(x_1, \ldots, x_k, x) \in \mathcal{S} &\Leftrightarrow \forall t \leq x,\ \exists z_1, \ldots, z_n : P(t, x_1, \ldots, x_k, z_1, \ldots, z_n) = 0 \\ &\Leftrightarrow \exists y : \forall t \leq x,\ \exists z_1, \ldots, z_n \leq y : P(t, x_1, \ldots, x_k, z_1, \ldots, z_n) = 0\end{aligned}$$

et ce dans la mesure où la borne y peut être choisie comme le plus grand de $n(x+1)$ entiers. On termine la preuve en appliquant les lemmes 3 et 2.

Vient maintenant la dernière[1] étape de cette démonstration. On va construire une machine de Turing qui va nous mener au résultat. Soit $\mathcal{P}$ une relation k-aire et soit T une machine de Turing, munie d'un ensemble d'états $q_0, \ldots, q_\nu$, qui, ayant démarré à q_0 sur le symbole le plus à gauche de $\ulcorner x_1 \urcorner\, 0\, \ulcorner x_2 \urcorner\, 0 \ldots 0\, \ulcorner x_k \urcorner\, 0$, s'arrête si et seulement si $(x_1, \ldots, x_k) \in \mathcal{R}$. Soit T' la machine obtenue en ajoutant à T : $(i, \alpha) \mapsto (\alpha, \nu+1, \blacktriangleright)$ où $i \in [\![0, \nu]\!]$ et la fonction de transition de T n'est pas définie pour (i, α). On a alors que $(x_1, \ldots, x_k) \in \mathcal{R}$ si et seulement si T' entre dans l'état $q_{\nu+1}$ (et s'arrête), ayant démarré à q_0 sur le symbole le plus à gauche de $\ulcorner x_1 \urcorner\, 0\, \ulcorner x_2 \urcorner\, 0 \ldots 0\, \ulcorner x_k \urcorner\, 0$.

On décrira les configurations de T' par des triplets $(l, n, r) \in \mathbb{N}^3$, où n est le numéro de l'instruction sur le point d'être exécutée, l est l'entier naturel dont la représentation en binaire est la séquence de 0 et de 1 sur la bande à gauche de la case pointée et r est l'entier naturel dont la représentation en binaire est la séquence de 0 et de 1, lue de droite à gauche, sur la bande à droite de la case pointée. La configuration initiale de T' est donc le triplet : $(0, 0, f_k(x_1, \ldots, x_k))$, f_k étant la fonction convenable qui code $(x_1, \ldots, x_k)$,

$$f : \begin{cases} f_1(x) = 2^{x+1} - 1 \\ f_{k+1}(x_1, \ldots, x_k) = 2 f_k(x_2, \ldots, x_k) + f_1(x_1). \end{cases}$$

On vérifie par récurrence que la relation $y = f_k(x_1, \ldots, x_k)$ est diophantienne pour tout k fixé.

Pour Q, couple antécédent/image de la fonction de transition de T', on note $\mathcal{S}_Q$ l'ensemble des sextuplets (l, n, r, l', n', r') tels que Q fait passer de (l, n, r) à (l', n', r').

Lemme 4 *$\mathcal{S}_Q$ est diophantienne pour tout Q fixé.*

Démonstration : si Q est de la forme $(i, \alpha) \mapsto (\beta, j, \blacktriangleright)$ alors :

$$(l, n, r, l', n', r') \in \mathcal{S}_Q \Leftrightarrow n = i \wedge n' = j \wedge l' = 2l + \beta \wedge r = 2r' + \alpha$$

1. ouf !

et si Q est de la forme $(i,\alpha) \mapsto (\beta, j, \blacktriangleleft)$ alors :

$$\begin{aligned}
&(l,n,r,l',n',r') \in \mathcal{S}_Q \\
&\Leftrightarrow n = i \wedge n' = j \vee \quad [(l = 2l' \wedge r' = 2(r+\beta-\alpha)) \\
&\qquad\qquad\qquad\qquad\quad \vee (l = 2l'+1 \wedge r' = 2(r+\beta-\alpha)+1)].
\end{aligned}$$

Cqfd.

Soit $Q_1, \ldots, Q_\mu$ une suite de tous les couples antécédents/images de la fonction de transition de T' et soit $\mathcal{S} = \bigcup_{i=1}^{\mu} \mathcal{S}_{Q_i}$. Alors $\mathcal{S}$ est une relation diophantienne.

Un calcul de T' est désormais assimilable à une suite de configurations (l_i, n_i, r_i) avec $i \in [\![0, k+1]\!]$ et telles que $(l_i, n_i, r_i, l_{i+1}, n_{i+1}, r_{i+1}) \in \mathcal{S}$ avec $i \in [\![0, k]\!]$ et $n_{k+1} = \nu + 1$. Dans une base B convenable (suffisamment grande), on peut poser :

$$L = \sum_{i=0}^{k} l_i B^i, \quad N = \sum_{i=0}^{k} n_i B^i, \quad R = \sum_{i=0}^{k} r_i B^i.$$

Grâce à la fonction Φ du lemme 2 on a que

$$\begin{aligned}
(x_1, \ldots, x_k) \in \quad & \mathcal{R} \Leftrightarrow \\
\exists B, L, N, R, k : \quad & \Phi(0,B,L) = 0 \wedge \Phi(0,B,N) = 0 \wedge \\
& \Phi(0,B,R) = f_k(x_1, \ldots, x_k) \wedge \Phi(k+1,B,N) = \nu + 1 \wedge \\
& \ \forall i \leq k : \quad (\Phi(i,B,L), \Phi(i,B,N), \Phi(i,B,R), \\
& \qquad\qquad\quad \Phi(i+1,B,L), \Phi(i+1,B,N), \\
& \qquad\qquad\quad \Phi(i+1,B,R)) \in \mathcal{S}.
\end{aligned}$$

Par le théorème 3, il vient que $\mathcal{R}$ est diophantienne. Le théorème 2 est ainsi démontré.

L'application du théorème de Matijasevič-Robinson au problème de Hilbert se fait à partir des deux constatations suivantes :

- a) *un ensemble D d'entiers positifs est récursif si et seulement si cet ensemble et son complémentaire sont récursivement énumérables,*
- b) *il existe un ensemble récursivement énumérable mais non récursif.*

La preuve de a) est simple : on peut facilement construire une machine de Turing qui s'arrête sur chaque entrée en composant les machines qui reconnaissent respectivement D et $\overline{D}$. Pour prouver b), on numérote l'ensemble dénombrable des algorithmes $\mathcal{P}_0, \mathcal{P}_1, \ldots$ Cette numérotation est une bijection de $\mathbb{N}$ dans l'ensemble des algorithmes. Soit $S(\mathcal{P}) = \{n : \mathcal{P}(n)$ répond « oui »$\}$ qui est un ensemble récursivement énumérable. Considérons maintenant $E = \{n : n \in S(\mathcal{P}_n)\}$. Cet ensemble est récursivement énumérable mais pas récursif car on aurait alors $\overline{E}$ récursivement énumérable et donc il existerait un entier k tel que $\overline{E} = S(\mathcal{P}_k)$. Mézalor si $k \in \overline{E}$ on a $k \in S(\mathcal{P}_k)$ et donc $k \in E$ et il vient une contradiction, si $k \in E$ alors $k \in S(\mathcal{P}_k) = \overline{E}$ et on a une contradiction. E n'est donc pas récursif.

Prenons une relation récursivement énumérable (donc diophantienne) mais non-récursive, appliquons une méthode générale de résolution à son polynôme associé, alors cette relation est récursive. On a bien montré une réponse négative au dixième problème de Hilbert.

2.1.2 Définition d'un ordre sur les polynômes

La définition exposée est celle de [2].
Pour donner une généralisation du fait que $XY > Y$ si $X > 1$, nous allons restreindre le domaine de chaque indéterminée à $\mathbb{N} - \{0, 1\}$. Les fonctions polynomiales seront donc de la forme : $(\mathbb{N} - \{0, 1\})^n \to \mathbb{N}$ où n est le nombre d'indéterminées du polynôme considéré. Les interprétations pouvant être obtenues par composition de polynômes, il convient de vérifier la stabilité de cette opération. La contrainte ajoutée est donc de n'avoir pour interprétations que des fonctions de $(\mathbb{N} - \{0, 1\})^n \to \mathbb{N} - \{0, 1\}$. Cette contrainte sera respectée si $\forall 1 \leq i \leq n,\ X_i > 1 \Rightarrow [\![f]\!](X_1, \ldots, X_n) > 1$, elle sera également vérifiée si $\forall (a_1, \ldots, a_n) \in (\mathbb{N} - \{0, 1\})^n,\ \forall 1 \leq i \leq n,\ [\![f]\!](a_1, \ldots, a_n) > a_i$. Vérifier que les coefficients des interprétations sont des entiers naturels est bien-sûr suffisant mais pas du tout nécessaire[2] dans la mesure où nombre de polynômes d'interprétation sont à coefficients dans $\mathbb{Z}$.

Comme dans l'introduction, nous notons $\succeq$ l'ordre sur les fonctions polynômes et définissons :

$$P \succeq Q \Leftrightarrow \forall (a_1, \ldots, a_n) \in (\mathbb{N} - \{0, 1\})^n,\ P(a_1, \ldots, a_n) \geq Q(a_1, \ldots, a_n),$$

Cet ordre n'est évidement pas total.

2.2 Méthodes d'ordonnancement des polynômes

La principale démarche pour tester si un polynôme est plus grand qu'un autre consiste en fait à tester la « positivité » de la différence, c'est-à-dire la comparer au polynôme nul. Les deux procédés qui suivent ont pour point commun de chercher à minorer cette différence par des polynômes dont on sait qu'ils sont « positifs » (car, par exemple, ils vérifient une condition suffisante, même très forte). Ils se distinguent essentiellement par leur façon de minorer un polynôme.

L'algorithme est le suivant, soit positif(P) la procédure qui teste si un polynôme est positif :

Positif(P) = Si tout coefficient de P est positif
    alors vrai

2. voir notamment le cas des λ-calculs avec substitutions explicites.

sinon si (on peut tranformer[3] P en Q) et (positif(Q))
alors vrai
sinon : on ne sait pas conclure.

La terminaison sera assurée si la procédure de transformation fait décroître le polynôme en un certain sens. Voyons maintenant deux exemples de transformation.

2.2.1 Une méthode d'Ahlem Ben Cherifa et Pierre Lescanne

Nous décrivons ici le premier des deux procédés exposés dans [2], tous deux implantés dans REVE et ORME. Nous écrirons les polynômes sous la forme :

$$P_i = \sum_{(p_1,\dots,p_m)\in\mathbb{N}^m} a^{[i]}_{p_1\dots p_m} X_1^{p_1}\dots X_m^{p_m}.$$

Principe et algorithme. On va prouver qu'un polynôme P est (éventuellement) positif en construisant une suite $P = P_0 \succeq P_1 \succeq \dots \succeq P_n \succeq 0$ où on sait que P_n est positif car tous ses coefficients sont positifs ou nuls. À chaque pas le polynôme P_i est vu comme $P_{i+1} + Q_i$ où Q_i est positif.

Le problème est de déterminer un polynôme P_{i+1} à partir de P_i. On va chercher à compenser les monômes négatifs. Soit

$$P = bX_1^{p_1}\dots X_m^{p_m} + aX_1^{q_1}\dots X_m^{q_m} + Q \text{ où } \begin{cases} b > 0, \\ a < 0, \\ \forall i, p_i \geq q_i \& \exists i_0 / p_{i_0} > q_{i_0} \end{cases}$$

$\mathbf{b} > -\mathbf{a}$ alors $P \succeq (b+a)X_1^{p_1}\dots X_m^{p_m} + Q$. En effet, si on soustrait P à cette quantité, il reste $a(X_1^{p_1}\dots X_m^{p_m} - X_1^{q_1}\dots X_m^{q_m})$ dont le premier terme est négatif et le second positif par définition.

$\mathbf{b} \leq -\mathbf{a}$ alors $P \succeq (b+a)X_1^{q_1}\dots X_m^{q_m} + Q$. En effet, si on soustrait P à cette quantité, il reste $b(X_1^{q_1}\dots X_m^{q_m} - X_1^{p_1}\dots X_m^{p_m})$ dont le premier terme est positif et le second négatif.

On définit ainsi une première version de la procédure « transforme » c'est-à-dire le passage de P_i à P_{i+1} en fonction des monômes M_1 et M_2.

Preuve de terminaison. Il faut maintenant vérifier que cette procédure termine toujours. La preuve se fait en définissant comme complexité : $c(P)$ = multi-ensemble des complexités des monômes, où $c(M_{p_1\dots p_m}) = c(a\prod_i X_i^{p_i})$ $= (\sum_i p_i, |a|)$ avec $a < 0$.

Il est clair que dans le premier cas ($b > -a$) on remplace par un monôme positif deux monômes dont un négatif, donc la complexité décroît. Quant

3. par une procédure « transforme ».

au second cas, on remplace deux monômes dont un négatif par un monôme négatif de même degré et de coefficient $(a+b)$. Comme $b \leq -a$, $|(a+b)| < |a|$ et la complexité décroît. On a bien prouvé que la procédure terminait.

2.2.2 Une méthode de Claude Marché

Ce procédé reprend les grandes idées du précédent tout en affinant les minorations. Il est donc plus puissant et est implanté dans CiME.

Principe et algorithme. Les polynômes sont toujours de la forme :

$$P_i = \sum_{(p_1,\dots,p_m)\in\mathbb{N}^m} a^{[i]}_{p_1\dots p_m} X_1^{p_m}\dots X_m^{p_m}.$$

L'idée est encore de bâtir une suite décroissante de polynômes dont le plus petit est positif ou nul. On va chercher à compenser le monôme négatif de plus haut degré total mais cette fois la minoration sera plus fine car on va utiliser le fait que $X_1^{q_1}\dots X_m^{q_m} \succ X_1^{q_1}\dots X_m^{q_m} - (X_1-1)^{q_m}\dots(X_m-1)^{q_m}$ (et non plus 0). On a donc :

$\mathbf{b} > -\mathbf{a}$ alors il vient :
$P \succeq (b+a)(\prod_i X_i^{p_i}) - a(\prod_i X_i^{q_i} - \prod_i (X_i-1)^{q_i})(\prod_i X_i^{p_i-q_i} - 1) + Q$
$= bX_1^{p_m}\dots X_m^{p_m} + aX_1^{q_m}\dots X_m^{q_m} + a(\prod_i (X_i-1)^{q_i}((\prod_i X_i^{p_i-q_i})-1)+Q,$

$\mathbf{b} \geq -\mathbf{a}$ alors, en effectuant un calcul similaire, on trouve :
$P \succeq b(\prod_i X_i^{p_i}) + a(\prod_i X_i^{q_i}) - a(\prod_i (X_i-1)^{q_i}((\prod_i X_i^{p_i-q_i}) - 1) + Q.$

Il suffit maintenant d'utiliser l'algorithme avec cette nouvelle procédure « transforme ».

Preuve de terminaison. On garde la même définition de la complexité que précédemment, c'est-à-dire : $c(P) = c(M_{p_1\dots p_m})$ (multi-ensemble) où $c(M_{p_1\dots p_m}) = c(aX_1^{p_1}\dots X_m^{p_m}) = (\sum_i p_i, |a|)$ avec $a < 0$. Dans le cas où $(a+b) > 0$ on minore $X_1^{p_m}\dots X_m^{p_m}$ par $\prod_i X_i^{q_i} - \prod_i (X_i-1)^{q_i}$ dont les degrés de tous les monômes à coefficient négatif sont strictement inférieurs à $\sum p_i$, on fait donc baisser la complexité. Dans le cas où $(a+b) \leq 0$ les monômes négatifs de plus haut degré qui apparaissent sont de la forme $bX_1^{q_m}\dots X_m^{q_m}$ et, comme $|a+b| < |a|$, la complexité décroît.

La minoration étant plus fine, on obtient de meilleurs résultats avec cette seconde méthode : par exemple si le polynôme $XY - X - Y + 1$ est bien annoncé positif par le procédé de C. Marché, la méthode d'A. Ben Cherifa et P. Lescanne ne permet pas de conclure. Cet algorithme ne permet néanmoins pas de prouver la « positivité » de $X^2 + Y^2 - 2XY + 1$.

Chapitre 3

Implantation dans C*i*ME

Cette implantation d'une méthode d'ordonnancement des polynômes a été réalisée en CAML-light 7.0, un langage de la famille ML développé et disponible *via* `ftp` à l'INRIA Rocquencourt.

Il s'agit de donner à C*i*ME la possibilité d'utiliser des ordres polynomiaux.

3.1 Le système C*i*ME

C*i*ME, Complet*i*on Modulo E, est une implantation, réalisée par Evelyne Contejean et Claude Marché, de l'algorithme de complétion normalisée dû à Claude Marché [14]. Cet algorithme permet de traiter, entre autres, des axiomes tels que AC + unité, AC + idempotence ou encore la théorie des groupes abéliens ou des corps finis...

Il n'est pas question ici de donner une description complète ou précise de C*i*ME, on pourra consulter la documentation [3] qu'en ont fait les auteurs.

Principe. En réécriture normalisée, les termes sont normalisés suivant un système de réécriture convergent équivalent à la théorie considérée. Le programme offre deux aspects de cette approche : on peut directement déclarer une des théories proposées par C*i*ME, les déductions utiliseront alors des connaissances à son sujet, on peut aussi déclarer la théorie à l'aide de laquelle l'unification sera effectuée.

Syntaxe. Les fichiers passés en argument se présentent comme une succession de déclarations conclue par le mot-clef `end`. L'utilisateur commence par déclarer les opérateurs avec leur arité et, le cas échéant, leurs propriétés (commutativité, associativité). Il peut alors éventuellement déclarer une théorie, des axiomes, puis un ordre, etc. Le travail présenté ici consiste à implanter un nouveau type d'ordre disponible, à savoir l'ordre polynomial.

On trouvera alors comme utilisable sous le mot-clef `order` la syntaxe :

`poly(`*Interprétation ; ...* `)`

où l'interprétation de f, opérateur d'arité n est :

`[`f`](x1,x2,...,xn)=`*polynôme en* $x_1, \ldots, x_n$

Les indéterminées du polynôme n'ayant rien à voir avec les variables des axiomes, les mêmes symboles pourront être utilisés.

3.2 Structure du programme

Savoir traiter les ordres polynomiaux impose, en premier lieu, de pouvoir manipuler des polynômes (en particulier de les définir et de savoir les manipuler) et, en second lieu, d'être capable d'appliquer une méthode d'ordonnancement. C'est donc tout naturellement que nous allons diviser notre travail en trois parties distinctes :

1. définitions, constructions et tests sur les polynômes,
2. opérations sur les polynômes,
3. calcul des interprétations et ordonnancement des polynômes.

Dans la mesure où ce programme doit prendre place au sein du système C*i*ME, il faudra modifier certaines de ses composantes, notamment les modules d'analyse des fichiers passés en argument.

3.3 Le module `poly`

C'est le module de base de l'implantation : ici sont définis les polynômes, les opérations de construction et les tests, plus généralement les opérations qui font intervenir la structure du polynôme.

`poly` est un module indépendant qui sera utilisé par tous les modules manipulant des polynômes : entre autres, le module de calcul et le module d'ordonnancement.

Analyse. Il faut définir ce qu'est un polynôme, on aura certainement besoin de monômes et de fonctions qui permettent de les considérer. Les quelques tests ne posent pas de problème d'analyse.

Structures et fonctions. La vision qu'on a habituellement d'un polynôme est qu'il est nul ou constitué de monômes. De la même manière, on voit un monôme comme un coefficient et une suite d'indéterminées élevées à diverses puissances. Nous allons donc définir le type monôme comme un enregistrement qui comprend le coefficient et la suite des indéterminées à leurs puissances respectives. Il est inutile de garder le « nom » des indéterminées pour peu qu'on veuille bien toujours les considérer dans le même ordre, une liste des puissances suffit alors à caractériser le monôme. Nous pouvons maintenant fixer le type monôme :

```
type monome = { coef : int; degres : int list };;
```

Un polynôme sera alors une liste de monômes de type polynôme, vide elle représentera le polynôme nul.

```
type polynome = Poly of monome list ;;
let polynome_nul = Poly [] ;;
```

Nous devons être capables de fabriquer un monôme et un polynôme, d'extraire un monôme d'un polynôme, d'exprimer son coefficient, sa liste des puissances et certaines d'entre elles, enfin nous devons être en mesure de connaître la dimension d'un polynôme (son nombre d'indéterminées). Ce qui nous donne :

```
(*coef  puissances -> monome*)
value fabrique_monome : int -> int list -> monome;;
value fabrique_polynome : monome -> polynome -> polynome ;;

(*premier monome du polynome*)
value premier_monome : polynome -> monome ;;
(*poly sans premier monome*)
value reste_polynome : polynome -> polynome ;;

(* coef de l'argument *)
value coef_monome : monome -> int ;;
(* liste des puissances de l'argument *)
value vars_monome : monome -> int list ;;
(* premier degre de la liste des puissances *)
value premier_degre : 'a list -> 'a ;;
(* reste de la liste des puissances *)
value reste_degres : 'a list -> 'a list ;;

(* espace de l'argument *)
value espace_polynome : polynome -> int ;;
```

En ce qui concerne les tests, il est utile de tester si un polynôme est le polynôme nul. Nous aurons besoin de certains critères sur les polynômes, en particulier seuls certains polynômes peuvent représenter des opérateurs AC et les polynômes constants doivent être supérieurs à 1 : pour éviter les erreurs, il faudra tester le respect de ces critères. Enfin, pour nous mettre à l'abri des oublis de déclaration de variables, nous vérifierons que toutes les indéterminées ont un sens.

```
(* teste si l'argument est le polynome nul *)
value polynome_est_nul : polynome -> bool ;;
(* teste si l'arg est constant et s'il est > 1 *)
value poly_gt_1 : polynome -> bool ;;
(* teste si l'arg represente un operateur AC *)
value ac_poly : polynome -> bool ;;
(* teste la presence de variable *)
value exist_var : monome -> bool ;;
```

Il reste à permettre l'affichage d'un polynôme, ie : le convertir en chaîne de caractères.

```
(* string_of_monome *)
value ecrit_monome : monome -> string ;;
(* string_of_polynome *)
value ecrit_polynome : polynome -> string ;;
```

3.4 Le module `poly_op`

Nous devons pouvoir effectuer des calculs sur des polynômes, `poly_op` va s'en charger. Ce module contient essentiellement les fonctions de base qui manipulent des polynômes, sans argument de structure.

Définissant des fonctions de base sur les polynômes, `poly_op` a besoin de `poly` et est utilisé par le module d'ordonnancement. Il peut servir à tout module manipulant des polynômes.

Analyse. Les opérations élémentaires sur les polynômes sont : l'addition et la multiplication. Elles permettent de construire l'élévation de puissance et la composition. Mais tout d'abord une constatation : compte tenu de la structure choisie, où la liste des puissances ne permet pas l'identification univoque d'un monôme, nous devrons, pour être efficaces, rassembler tous les monômes de même degré.

Si les opérations « arithmétiques » ne posent pas de problème, les fonctions de substitutions sont plus délicates à mettre en œuvre. Elles sont rendues nécessaires par le principe même de l'interprétation, à base de composition d'interprétations. Pratiquement, pour chaque monôme où la substitution doit avoir lieu, on élève à la puissance désirée le polynôme de remplacement puis on effectue le produit avec le monôme où X_n (indéterminée à remplacer) est affecté de la puissance 0. On doit pouvoir effectuer cette substitution pour toutes les indéterminées et tout le polynôme.

Structures et fonctions. Le rassemblement des monômes de même degré est fait par la fonction `propre`, invisible à l'extérieur du module. Nous pouvons maintenant additionner et multiplier en toute sécurité.

```
value polysum : polynome -> polynome -> polynome ;;
value polyprod : polynome -> polynome -> polynome ;;
value polyminus : polynome -> polynome ;;
value polypow : polynome -> int -> polynome ;;
```

Le passage à l'opposé est obtenu par multiplication avec le polynôme (-1), on a donc eu besoin de créer ce polynôme. Dans le même ordre d'idées, nous créerons des polynômes réduits à une indéterminée choisie.

```
value polycst :  int -> int -> polynome;;
value polyvar :  int -> int -> polynome;;
```

Il n'est pas en contradiction avec nos principes de mettre ces fonctions dans `poly_op` : l'utilisateur n'a pas à passer en argument des éléments structurels du type `polynome`.

Pour effectuer la substitution d'une indéterminée choisie par un polynôme, nous disposons de la fonction :

```
value polysubst : int -> polynome -> polynome -> polynome ;;
```

qui remplace dans le troisième argument toutes les occurrences de X_n (où n est le premier argument) par le deuxième. Passons au cas général :

```
value substituer_polynomes :
int -> polynome list -> polynome -> polynome ;;
(* (substituer_polynomes n list_p p)
remplace dans p (de Z[X1,_,Xm]) chaque occurrence
 des variables X_i par le ième élément de list_p
qui est une liste de polynomes de Z[X1,...,Xn].*)
```

Si on voit P et les éléments Q_i de `list_p` comme des fonctions alors :

$$\begin{array}{l} P : \mathbb{Z}[X_1, \ldots, X_m] \to \mathbb{Z} \\ Q_i : \mathbb{Z}[X_1, \ldots, X_n] \to \mathbb{Z} \\ n = \dim(\mathbb{Z}[X_1, \ldots, X_n]) \end{array}$$

Cette fonction est récursive sur le polynôme, une récursion sur la liste ne donnerait pas un résultat correct.

3.5 Le module `ord_poly`

Le rôle de ce module est de donner, si possible, une réponse à la question de l'ordre entre deux polynômes : il représente l'implantation d'une méthode d'ordonnancement.

`ord_poly` utilise, bien-sûr, `poly` et `poly_op` son implantation dans C*i*ME impose le respect de certains types prédéfinis et donc le recours à d'autres modules [1]. Il peut être exploité par des modules utilisant les ordres polynomiaux.

Analyse. Il faut ici que sur l'entrée d'une fonction d'interprétation $[\![\cdot]\!]$ on obtienne une fonction de comparaison qui ordonne les termes s et t suivant une relation d'ordre sur $[\![s]\!]$ et $[\![t]\!]$. Les interprétations étant polynomiales, nous avons choisi de les ordonner suivant la méthode de C. Marché, plus puissante que celle de P. Lescanne et A. Ben Cherifa pour une difficulté d'implantation similaire. Puisque l'algorithme décrit précédemment teste la « positivité » d'un polynôme, il faudra effectuer plusieurs tests et déduire le résultat de leur combinaison.

Structures et fonctions. L'algorithme de positivité est déjà décrit, il reste néanmoins le problème du choix des monômes à compenser. Nous avons pris le parti, au vu des tailles généralement modestes des interprétations, de tester toutes les compensations jusqu'à épuisement des possibilités (échec) ou obtention de la positivité (réussite). De plus, en raison de l'aspect non-comparable de certains polynômes, il faut séparer les cas de positivité stricte et large. Les fonctions `poly_positif_strict` et `poly_positif_ou_nul` n'en comportent pas moins la même structure : on essaye de séparer le polynôme différence en monômes positifs et négatifs, on parcourt la liste de ces derniers en appliquant le critère retenu. Ces fonctions renvoient des booléens et sont donc combinées dans une fonction visible qui, elle, retourne un objet du type `resultat_comparaison`, type prédéfini du système C*i*ME.

```
value compare_poly :
            polynome -> polynome -> comparison_result ;;
```

Une fois définie une procédure `interprete` qui compose les résultats de $[\![\cdot]\!]$ pour donner une interprétation d'un terme, on est capable de construire une fonction `poly_ord` appelant `interprete` et `compare_poly` sur de bons arguments.

```
value poly_ord : (symbol_id -> polynome) -> term ordering ;;
```

On vérifie enfin que la fonction d'interprétation gère correctement le cas des opérateurs AC intervenant sur plus de deux termes.

1. `orderings`, `symbols` et `terms`. Voir section suivante.

3.6 L'insertion dans C*i*ME

L'implantation que nous venons de décrire dans ses grandes lignes prend place dans le système C*i*ME. Pour pouvoir l'exploiter, il a fallu avoir recours et modifier certains modules dudit système.

Recours : types et fonctions prédéfinis. C'est principalement dans le module `ord_poly` que des appels à des types et fonctions extérieurs aux modules d'ordres polynomiaux sont effectués. La généralité des types d'ordre retournés (`compare_poly`), les fonctions sur la structure des termes (`interprete`, `poly_ord`) et les tests sur les symboles de certains de ces termes (`interprete`) sont autant de raisons d'utiliser les modules `orderings` `terms` et `symbols`.

Mise à jour des analyses lexicale et syntaxique. Il nous faut savoir traiter les fichiers d'arguments qui requièrent les ordres polynomiaux.

Lexing. On ajoute le mot-clef `poly` déclarant que l'ordre polynomial est demandé. On définit un nouvel état du lexer qui lui permet de renvoyer les tokens opportuns lors de l'analyse d'un polynôme. On fixe enfin le format d'un polynôme : les indéterminées seront de la forme x ou xn c'est-à-dire lettre ou lettre suivie d'un nombre. Deux tokens associés à « [» et «] » sont également introduits.

Parsing. Un ordre peut maintenant être du type :

```
keyword_poly PARGAUCHE poly_decl_list PARDROITE
    { POLY($3) }
```

où est renvoyée la liste des déclarations, à savoir :

```
poly_decl_list:
  poly_decl                     { [$1] }
| poly_decl POINT_VIRGULE poly_decl_list{ $1::$3 };
```

Vient ensuite la description de la structure d'un polynôme comprenant des tests pour éviter quelques étourderies.

```
poly_decl:
  entete_poly EGAL poly...
```

On y teste si le polynôme est une constante supérieure à 1 ou si, déclaré AC, il en vérifie bien les conditions nécessaires.

```
entete_poly:
  CROGAUCHE poly_ident CRODROIT
```

```
    { current_poly_vars := [];
      $2 }
| CROGAUCHE poly_ident CRODROIT PARGAUCHE poly_var_list_end
    { current_poly_vars := $5...
```

On teste alors le respect de l'arité déclarée de l'opérateur représenté et la distinction des symboles d'indéterminées.

```
poly_ident:
  IDENT { etat_analyse.lexer_state <- Poly_lexer;
          (get_symbol_id $1) };

poly_var_list_end:
  POLY_VAR PARDROITE                  { [$1] }
| POLY_VAR VIRGULE poly_var_list_end  { $1::$3 };
```

Et enfin

```
poly:
  POLY_VAR
    { let m = monome_de_variable $1
      in
      if  (exist_var m)
      then fabrique_polynome (monome_de_variable $1) polynome_nul
      else semantical_error "Undeclared polynomial variable"
      }
| INT
    { polycst $1 (list_length !current_poly_vars)
      }
| PARGAUCHE poly PARDROITE
    { $2 }
| poly PLUS poly
    { polysum $1 $3 }
| poly MINUS poly
    { polysum $1 (polyminus $3) }
| poly MULT poly
    { polyprod $1 $3 }
| poly EXP INT
    { polypow $1 $3 };
```

où on vérifie que dans le cas `POLY_VAR` les indéterminées sont bien déclarées.

Autres modifications. Les quelques modifications qui restent à effectuer ne sont que l'ajout des appels aux nouvelles fonctions dans les cas de recours à l'ordre polynomial.

3.7 Fichiers d'exemples

Les tests définis précédemment et la correction des nouvelles fonctionnalités sont mis à l'épreuve par chacun des fichiers tests suivants, disponibles dans `cime/examples/term_orderings/poly` .

Erreurs. Mise en œuvre des tests, le passage de ces fichiers en argument provoque l'affichage des messages d'erreur correspondants.

`illegal1` vérifie la distinction des indéterminées déclarées.

`illegal2` teste la déclaration de toutes les indéterminées.

`illegal3`... contrôlent respectivement les arités explicites et implicites.

`illegal4`... mettent à l'épreuve le respect des conditions nécessaires AC, respectivement : la symétrie des termes de degré 1, les relations sur les coefficients et le degré maximal par indéterminée.

`illegal5` impose la valeur minimale des polynômes constants.

Terminaisons. Correction de l'algorithme.

`test_poly` où la complétion échoue car l'orientation de la distributivité est indécidable.

`test_poly2` addition des entiers naturels.

`test_poly3` addition et multiplication des entiers naturels.

`test_poly4` anneaux commutatifs, méthode directe mais les interprétations sont difficiles à trouver.

`test_poly5` anneaux commutatifs, avec utilisation de mapo et interprétations plus simples.

`test_poly6` bonne prise en compte de la composition d'opérateurs AC dans des termes de type $a + b + c$ (calcul).

`test_poly7` bonne prise en compte de la composition d'opérateurs AC dans des termes de type $a + b + c$ (variables).

Chapitre 4

λ-calculs avec substitutions explicites

Les λ-calculs avec substitutions explicites sont des systèmes de réécriture du premier ordre qui permettent, en précisant de manière très fine le processus de substitution, de décrire agréablement mais surtout uniformément les implantations de λ-calculs et, donc, de langages de programmation fonctionnelle. Tout au long de ce chapitre, nous expliquerons pourquoi ces λ-calculs sont apparus, nous détaillerons la structure de certains d'entre eux, nous verrons enfin, à travers quelques exemples, dans quelle mesure les nouvelles capacités de CiME lui autorisent le traitement des λ-calculs avec substitutions explicites.

4.1 λ-calcul et réécriture

Limites du λ-calcul classique. La β-réduction est le principal mécanisme du λ-calcul. On la définit habituellement par :

$$(\lambda x.E_1\ E_2) \xrightarrow{\beta} E_1[x \mapsto E_2]$$

où $[x \mapsto E_2]$ est la substitution de la variable x par le λ-terme E_1. Hélas, dans le λ-calcul classique, cette substitution ne peut pas être exprimée dans le cadre du calcul. Elle se tient hors d'icelui avec un formalisme externe et ne rentre donc pas dans le cadre de la réécriture. Ce handicap fit plaider Curry [4] en faveur de la logique combinatoire qui, elle, offre un traitement complet des substitution grâce à un système de réécriture du premier ordre où les substitutions sont bien exprimées. Néanmoins ce formalisme est bien moins naturel que le λ-calcul pour exprimer le concept de fonction. C'est pourquoi, en 1978, De Bruijn [8] exposa un système utilisant de nouvelles notations et proposant un traitement complet et correct des substitutions : le premier λ-calcul avec substitutions explicites. Ces calculs possèdent ainsi dans le même contexte la β-conversion et une description de l'évaluation du processus de substitution.

4.2 Notation de De Bruijn et substitutions explicites

4.2.1 La notation avec indices de De Bruijn

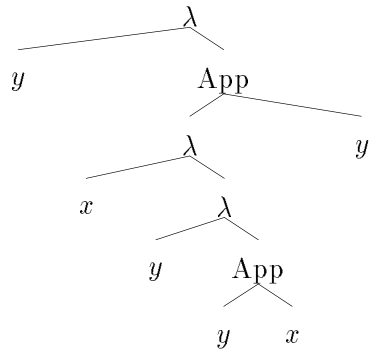
Indices et niveaux. Pour éviter l'α-conversion (c'est-à-dire le renommage dans les termes), De Bruijn [7] propose deux nouvelles notations :

- les indices, basés sur le nombre d'abstractions au-dessus de la variable concernée,
- les niveaux, basés sur une indexation canonique, invariante dans un λ-terme « pur » c'est-à-dire sans application de substitution.

La grande majorité des descriptions de λ-calculs avec substitutions explicites utilise les indices de De Bruijn ; les niveaux, rarement cités dans la littérature, n'interviennent que dans deux calculs : $\lambda\sigma$-calcul avec noms [1] et $\lambda\chi$ [13], nous ne détaillerons donc que les indices.

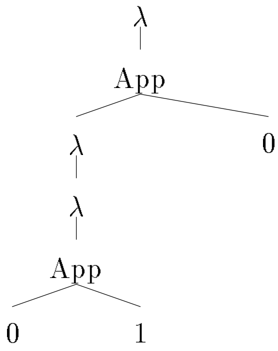
Les indices de De Bruijn. Introduite en 1972, cette notation permet de se passer du nom des variables en préférant indiquer à quel niveau d'abstraction elles sont liées. Considérons le λ-terme suivant : $\lambda y.(\lambda x.(\lambda y.(x\ y))\ y)$. En posant les conventions : App pour application, λ pour abstraction, variable pour variable, on peut l'écrire comme un graphe sous la forme :

λ-terme en arbre



Si on ne considère que les niveaux d'abstraction, ce graphe devient :

Arbre suivant De Bruijn



Et le terme de départ s'écrit donc : $\lambda(\lambda(\lambda(0\ 1))\ 0)$. On évite ainsi les problèmes de renommage et les risques de capture, l'explicitation des substitution est, de plus, rendue plus aisée car on va travailler sur les entiers.

4.2.2 Conventions d'écriture

Nous allons définir ici les notations de certains opérateurs utilisés dans la suite. Il convient de remarquer qu'ils ne seront pas tous utilisés dans chacun des λ-calculs décrits plus loin. De même, de nouveaux opérateurs spécifiques[1] peuvent apparaître. Il a semblé plus naturel de les spécifier au cours des constructions, là où leur utilité est évidente.

Opérations sur les termes. Les variables sont représentées par des indices de De Bruijn notés n... comme on le ferait pour des entiers positifs. Lorsque nous voudrons exprimer la variable indicée par n, nous utiliserons la notation $V(n)$. Il peut paraître curieux d'expliciter cet opérateur mais, pour prouver la terminaison de certains des λ-calculs, nous avons besoin d'une interprétation de celui-ci ; en outre, les entiers naturels interviennent ailleurs dans le calcul.

L'application de deux λ-termes a et b est notée : $(a\ b)$. De la même manière classique, λa désigne l'abstraction.

Nous écrirons le résultat de l'application d'une substitution σ à un λ-terme a sous la forme : $a[\sigma]$. Le détail de cet opérateur rejoint la principale motivation des λ-calculs avec substitutions explicites.

Opérations de substitutions. Les notations sont celles adoptées par Pierre Lescanne dans l'article qui nous sert de référence [12]. Pour clarifier, nous poserons que $S(n)$ est le successeur de n. Nous noterons id la substitution identité, $\uparrow$ la substitution constante telle que l'image de $V(n)$ est $V(S(n))$,

$$\uparrow :\ V(1) \mapsto V(2),\ \ldots,\ V(n) \mapsto V(S(n)).$$

1. notamment Φ et Ψ.

Remarquons que cette substitution permet de se débarrasser de n. En effet : $V(n) \equiv V(0)[\uparrow^n]$ pour $n > 1$.
Nous aurons besoin d'un opérateur de construction des substitutions qui, à partir d'un terme a et d'une substitution σ renvoie :

$$\begin{array}{rcl} a \cdot \sigma & : & V(1) \mapsto a, \\ & & V(2) \mapsto \sigma(V(1)), \\ & & \vdots \\ & & V(S(n)) \mapsto \sigma(V(n)) \end{array}$$

Enfin introduisons l'opérateur $\Uparrow$ qui à une substitution σ associe une substitution $\Uparrow(\sigma)$ telle que :

$$\begin{array}{rcl} \Uparrow(\sigma) & : & V(1) \mapsto V(1), \\ & & V(2) \mapsto \sigma(V(1))[\uparrow], \\ & & \vdots \\ & & V(S(n)) \mapsto \sigma(V(n))[\uparrow] \end{array}$$

4.3 Descriptions de quelques λ-calculs à substitutions explicites

Nous allons ici détailler les principes de construction et les idées directrices de quelques λ-calculs avec substitutions explicites. Nous verrons ainsi comment décrire les substitutions sans redondance, c'est-à-dire sans opérateur superflu, quitte à en introduire de plus judicieux.

La description à donner est bien sûr celle de l'opérateur d'application de substitution : $\_[\_]$. On le définit sur les variables et par ses propriétés de distributivité par rapport à l'application et à l'abstraction.

4.3.1 Construction de $\lambda\phi$

Deux principes vont nous guider dans cette construction :

– le calcul ϕ doit pouvoir réduire tout λ-terme qui contient une substitution,

– de nouveaux opérateurs ne seront introduits que s'il est impossible de décrire une forme réduite à partir de ceux qui existent déjà.

Les quatre opérations sur les termes disponibles au départ sont : App, $\lambda\_$, $V(\_)$ et $\_[\_]$. Nous allons devoir décrire la réduction des termes de la forme $a[\sigma]$ et donc donner une règle pour chaque forme possible de a. Le point de départ du raisonnement est la règle de réécriture de la β-réduction :

$$\beta \ : \ (\lambda a \ b) \longrightarrow a[b \cdot id].$$

Application. On a alors immédiatement la règle grâce à la distributivité :

$$(\mathrm{App}) \quad (a\ b)[\sigma] \longrightarrow (a[\sigma]\ b[\sigma]).$$

Abstraction. Pour réduire ce terme on doit pouvoir faire « passer » la substitution dans le λ, il faut donc la transformer. Quel est la modification des indices engendrée par cette opération ? Les images par substitution gagnent un niveau d'abstraction car elles passent sous un λ. Par définition de la notation en indices de De Bruijn, il faut incrémenter les indices des images. On introduit donc l'opérateur $\Uparrow$ déjà défini et on a :

$$(\mathrm{Abs}) \quad (\lambda a)[\sigma] \longrightarrow \lambda(a[\Uparrow(\sigma)]).$$

Application de substitution. Dans ce cas on applique une substitution τ à un terme de la forme $a[\sigma]$. Or, le premier principe de ϕ est de pouvoir réduire tout terme qui contient une substitution. Par récurrence ϕ doit donc réduire $a[\sigma][\tau]$ sans introduction d'une nouvelle règle.

Variable. C'est le cas le plus complexe : la variable pouvant être $V(1)$ ou $V(S(n))$ et la substitution le résultat d'une application de composition d'opérateurs. Nous allons traiter, pour chaque forme de la substitution σ, les deux cas de variable. Les dénominations des règles sont celles de [12].

- *id*. L'identité ne modifie pas les indices, on a donc :

$$(\mathrm{VarId}) \quad V(n)[id] \longrightarrow V(n).$$

- $a \cdot \tau$. Par définition de $\cdot$ la première image est a, les autres sont l'application de τ sur les variables représentées par l'indice précédent, soit :

$$\begin{array}{lrcl} (\mathrm{FVarCons}) & V(1)[a \cdot \tau] & \longrightarrow & a \\ (\mathrm{RVarCons}) & V(S(n))[a \cdot \tau] & \longrightarrow & V(n)[\tau]. \end{array}$$

- $\Uparrow(\tau)$. Par définition, $V(1)$ est invariant par $\Uparrow(\tau)$. Que se passe-t-il si la variable est de la forme $V(S(n))$? On devrait avoir pour résultat $(\tau V(n))[\uparrow]$ c'est-à-dire $V(n)[\tau \circ \uparrow]$ où $\circ$ est la composition de substitutions. On peut introduire un opérateur φ tel que $\varphi(\tau) = \tau \circ \uparrow$. Mézalor, on n'a aucun moyen de décrire la réduction du motif $V(n)[\varphi(\varphi(\tau))]$. C'est pourquoi on préfère à φ la notation $\Phi(\tau, p) = \tau \circ \uparrow^p$. Vérifions que les motifs comportant Φ sont réductibles :
  - $\Phi(\Phi(\tau, p), q)$. Par définition de la composition on doit avoir $(\tau \circ \uparrow^p) \circ \uparrow^q)$ donc $\tau \circ \uparrow^{p+q}$ soit :

$$(\mathrm{VarPhiPhi}) \quad V(n)[\Phi(\Phi(\tau, p), q)] \longrightarrow V(n)[\Phi(\tau, p+q)],$$

de plus :

$$\begin{array}{lrcl} \text{(Plus1)} & n+1 & \longrightarrow & S(n) \\ \text{(PlusS)} & n+S(m) & \longrightarrow & S(n+m). \end{array}$$

- $\Phi(id, p)$. En utilisant le fait que V et Φ admettent les mêmes entiers (d'où intérêt de la notation $V(n)$), on tire la règle :

$$\text{(VarPhiId)} \quad V(n)[\Phi(id,p)] \longrightarrow V(n+p).$$

- $\Phi(a \cdot \tau, p)$. On a alors les deux règles :

$$\begin{array}{lrcl} \text{(FVarPhiCons)} & V(1)[\Phi(a\cdot\tau,p)] & \longrightarrow & a[\Phi(id,p)] \\ \text{(RVarPhiCons)} & V(S(n))[\Phi(a\cdot\tau,p)] & \longrightarrow & V(n)[\Phi(\tau,p)]. \end{array}$$

- $\Phi(\Uparrow(\tau), p)$. C'est le dernier cas, on en tire :

$$\begin{array}{lrcl} \text{(FVarPhiLift)} & V(1)[\Phi(\Uparrow(\tau),p)] & \longrightarrow & V(S(p)) \\ \text{(RVarPhiLift)} & V(S(n))[\Phi(\Uparrow(\tau),p)] & \longrightarrow & V(n)[\Phi(\tau,S(p))]. \end{array}$$

Tous les cas ont été passés en revue, on peut alors exprimer $\Uparrow$:

$$\begin{array}{lrcl} \text{(FVarlift)} & V(1)[\Uparrow(\tau)] & \longrightarrow & V(1) \\ \text{(RVarlift)} & V(S(n))[\Uparrow(\tau)] & \longrightarrow & V(n)[\Phi(\tau,1)]. \end{array}$$

La preuve de terminaison de $\lambda\phi$ sur les termes clos sera effectuée par le système CiME lors du traitement de quelques exemples.

4.3.2 Construction de $\lambda\psi$

Une deuxième approche qui peut être faite consiste à introduire un nouvel opérateur dans la règle de réécriture de la β-réduction. C'est ce que fait Ríos [15] lorsqu'il présente $\lambda\psi$.

La règle de réécriture de la β-réduction devient :

$$\beta : \quad (\lambda a\ b) \longrightarrow a[b/].$$

L'opérateur / associe à un terme a une substitution σ telle que :

$$\begin{array}{rcl} \sigma & : & V(1) \mapsto a, \\ & & V(2) \mapsto V(1), \\ & & \vdots \\ & & V(S(n)) \mapsto V(n) \end{array}$$

On peut alors recommencer une construction systématique du calcul.

Les opérations sur les termes sont inchangées : on a toujours V, l'application, l'abstraction et l'application de substitution. Pour traiter / on dispose de :

$$\begin{array}{lrcl} \text{(FVar)} & V(1)[a/] & \longrightarrow & a \\ \text{(RVar)} & V(S(n))[a/] & \longrightarrow & V(n). \end{array}$$

On introduit pour les mêmes raisons que dans $\lambda\phi$ l'opérateur Φ. Si les règles restent valables, il faut néanmoins ajouter un cas dans l'étude de Φ : à savoir $\Phi(a/,p)$. On a alors :

$$\begin{array}{lrcl} \text{(FVarPhiSl)} & V(1)[\Phi(a/,p)] & \longrightarrow & a[\Psi(p)] \\ \text{(RVarPhiSl)} & V(S(n))[\Phi(a/,p)] & \longrightarrow & V(S(n+p)) \end{array}$$

où $\Psi(p)$ est une écriture de $\uparrow^p$. Cela amène encore le cas de la composition de Φ et Ψ,

$$\text{(VarPhiPsi)} \quad V(n)[\Phi(\Psi(q),p)] \quad \longrightarrow \quad V((n+q)+p)$$

et le cas d'application d'une substitution obtenue grâce à Ψ,

$$\text{(VarPsi)} \quad V(n)[\Psi(p)] \quad \longrightarrow \quad V(n+p).$$

La preuve de terminaison de $\lambda\psi$ sur les termes clos sera effectuée par le système C*i*ME lors du traitement de quelques exemples, les interprétations étant presque semblables à celles de $\lambda\phi$.

4.3.3 Construction de $\lambda\upsilon$

Nous décrivons ici un des plus simples λ-calculs avec substitutions explicites, le système $\lambda\upsilon$ de Pierre Lescanne [12] ne comporte en effet que huit règles et trois opérateurs de substitution.

Construction. On va réintroduire l'opérateur φ, on a donc :

$$\text{(RVarLift)} \quad V(S(n))[\Uparrow(\sigma)] \quad \longrightarrow \quad V(n)[\varphi(\sigma)].$$

En exprimant les réductions de φ on obtient la règle :

$$\text{(FVarphiSl)} \quad V(1)[\varphi(a/)] \quad \longrightarrow \quad a[\uparrow]$$

où, par définition,

$$\text{(VarShift)} \quad V(n)[\uparrow] \quad \longrightarrow \quad V(S(n)).$$

Pour réduire le nombre d'opérateurs, on va faire disparaître φ en préférant une nouvelle version de (RVarLift) :

$$\text{(RVarLift)} \quad V(S(n))[\Uparrow(\sigma)] \quad \longrightarrow \quad V(n)[\sigma][\uparrow].$$

On dispose alors d'un système très simple :

$$
\begin{array}{|lrcl|}
\hline
(\beta) & (\lambda a\ b) & \longrightarrow & a[b/] \\
 & & & \\
(\mathrm{App}) & (a\ b)[\sigma] & \longrightarrow & (a[\sigma]\ b[\sigma]) \\
(\mathrm{Abs}) & (\lambda a)[\sigma] & \longrightarrow & \lambda(a[\Uparrow(\sigma)]) \\
 & & & \\
(\mathrm{FVar}) & V(1)[a/] & \longrightarrow & a \\
(\mathrm{RVar}) & V(S(n))[a/] & \longrightarrow & V(n) \\
 & & & \\
(\mathrm{FVarLift}) & V(1)[\Uparrow(\sigma)] & \longrightarrow & V(1) \\
(\mathrm{RVarLift}) & V(S(n))[\Uparrow(\sigma)] & \longrightarrow & V(n)[\sigma][\uparrow] \\
 & & & \\
(\mathrm{VarShift}) & V(n)[\uparrow] & \longrightarrow & V(S(n)) \\
\hline
\end{array}
$$

Preuve de terminaison. C'est grâce à l'utilisation d'interprétations qu'on va pouvoir montrer la terminaison de λv. Le système d'interprétation est :

$$
\begin{array}{lcllcl}
[\![V(n)]\!]_1 & = & 2^{[\![n]\!]} & [\![V(n)]\!]_2 & = & 2^{[\![n]\!]} \\
[\![S(n)]\!]_1 & = & [\![n]\!]+1 & [\![S(n)]\!]_2 & = & [\![n]\!]+1 \\
[\![1]\!]_1 & = & 2 & [\![1]\!]_2 & = & 2 \\
[\![(a\ b)]\!]_1 & = & [\![a]\!]+[\![b]\!]+1 & [\![(a\ b)]\!]_2 & = & [\![a]\!]+[\![b]\!]+1 \\
[\![\lambda a]\!]_1 & = & [\![a]\!]+1 & [\![\lambda a]\!]_2 & = & [\![a]\!]+1 \\
[\![a[\sigma]]\!]_1 & = & [\![a]\!][\![\sigma]\!] & [\![a[\sigma]]\!]_2 & = & [\![a]\!][\![\sigma]\!] \\
[\![\Uparrow(\sigma)]\!]_1 & = & [\![\sigma]\!] & [\![\Uparrow(\sigma)]\!]_2 & = & [\![\sigma]\!]+1 \\
[\![\uparrow]\!]_1 & = & 2 & [\![\uparrow]\!]_2 & = & 3 \\
[\![a/]\!]_1 & = & \text{quelconque} & [\![a/]\!]_2 & = & \text{quelconque}
\end{array}
$$

On a ici une interprétation exponentielle (pour $V(n)$) qui n'est pas encore implantée dans CiME.

4.4 Quelques exemples traités par CiME

Nous allons maintenant prouver les terminaisons de $\lambda\phi$ et $\lambda\psi$ en utilisant des interprétations polynomiales. C'est en fait le système CiME qui, sur l'entrée des axiomes et des interprétations, va calculer la terminaison.

4.4.1 Terminaison de $\lambda\phi$

Interprétations. Dans le cas d'équivalence entre deux interprétations $[\![]\!]_1$, l'équation étant inorientable, on utilise un deuxième jeu $[\![]\!]_2$ pour résoudre le problème. Ces deux jeux sont :

$$
\begin{array}{lcllcl}
[\![V(n)]\!]_1 & = & [\![n]\!] & [\![V(n)]\!]_2 & = & [\![n]\!] \\
[\![S(n)]\!]_1 & = & [\![n]\!] & [\![S(n)]\!]_2 & = & [\![n]\!]+1 \\
[\![1]\!]_1 & = & 2 & [\![1]\!]_2 & = & 2 \\
[\![(a\ b)]\!]_1 & = & [\![a]\!]+[\![b]\!] & [\![(a\ b)]\!]_2 & = & [\![a]\!]+[\![b]\!]+1 \\
[\![\lambda a]\!]_1 & = & [\![a]\!]+1 & [\![\lambda a]\!]_2 & = & [\![a]\!] \\
[\![a[\sigma]]\!]_1 & = & [\![a]\!][\![\sigma]\!] & [\![a[\sigma]]\!]_2 & = & [\![a]\!]([\![\sigma]\!]+1) \\
[\![a\cdot\sigma]\!]_1 & = & [\![a]\!]+[\![\sigma]\!] & [\![a\cdot\sigma]\!]_2 & = & [\![a]\!]+[\![\sigma]\!] \\
[\![id]\!]_1 & = & 2 & [\![id]\!]_2 & = & 2 \\
[\![\Uparrow(\sigma)]\!]_1 & = & [\![\sigma]\!] & [\![\Uparrow(\sigma)]\!]_2 & = & 4[\![\sigma]\!] \\
[\![\Phi(\sigma,p)]\!]_1 & = & [\![\sigma]\!]([\![p]\!]-1) & [\![\Phi(\sigma,p)]\!]_2 & = & [\![\sigma]\!]([\![p]\!]+1) \\
[\![p+q]\!]_1 & = & [\![p]\!]+[\![q]\!]-2 & [\![p+q]\!]_2 & = & [\![p]\!]+2[\![q]\!]
\end{array}
$$

Arguments. Le fichier d'argument va donc comprendre la déclaration des opérateurs, les équations à orienter et l'ordre à utiliser. Puisqu'il y a deux interprétations on utilise une composition d'ordres, lexicographique et polynomial.

fichier d'arguments pour $\lambda\phi$

```
%
% System lambda-phi of Lescanne [From lambda-sigma to
%lambda-upsilon, a journey through calculi of explicit
%substitutions. POPL'94]
%
% termination using polynomial interpretations
%
% Xavier Urbain
%
% $Id: lambda_phi.cim,v 1.1 1996/07/16 12:00:33 urbain Exp $

operators
% # = lift, a&b = a[b]
  V,S,lambda,#  : unary

  1,id          : constant

  app,&,.,+     : infix binary
  phi           : binary

  a,b,s,n,p,q,m : variable

axioms
```

```
%Beta
%     lambda(a) app b = a&(b.id) ;

%App
          (a app b)&s = (a&s) app (b&s) ;
%Lambda
         lambda(a)&s  = lambda(a&#(s)) ;
%VarId
              V(n)&id = V(n);
%FVarCons
           V(1)&(a.s) = a ;
%RVarCons
        V(S(n))&(a.s) = V(n)&s ;
%FVarLift
            V(1)&#(s) = V(1) ;
%RVarLift
         V(S(n))&#(s) = V(n)&phi(s,1) ;
%VarPhiId
       V(n)&phi(id,p) = V(n+p) ;
%VarPhiPhi
 V(n)&phi(phi(s,q),p) = V(n)&phi(s,(q+p)) ;
%FVarPhiCons
    V(1)&phi((a.s),p) = a&phi(id,p) ;
%RVarPhiCons
 V(S(n))&phi((a.s),p) = V(n)&phi(s,p) ;
%FVarPhiLift
     V(1)&phi(#(s),p) = V(S(p)) ;
%RVarPhiLift
  V(S(n))&phi(#(s),p) = V(n)&phi(s,S(p)) ;

%Plus1
                  n+1 = S(n) ;
%PlusS
                n+S(m) = S(n+m) ;


order

  lexico(
    poly([V](n)=n; [S](n)=n; [1]=2; [app](a,b)=a+b;
[lambda](a)=a+1; [&](a,s)=a.s;[.](a,s)=a+s; [id]=2; [#](s)=s;
[phi](s,p)=s.(p-1);[+](p,q)=p+q-2);
    poly([V](n)=n; [S](n)=n+1; [1]=2; [app](a,b)=a+b+1;
[lambda](a)=a; [&](a,s)=a.(s+1); [.](a,s)=a+s; [id]=2;
```

```
[#](s)=4.s; [phi](s,p)=s.(p+1); [+](p,q)=p+2.q))

end
```

Résultat. C*i*ME oriente les règles de la façon suivante :

```
> cime cime/examples/explicit-subsitutions/lambda_phi
Welcome to CiME v1.11 - SunOS-5.4 - 07/18/96 16:50:22


New rule produced :  [1] a+1 -> S(a)
New rule produced :  [2] V(a) & id -> V(a)
New rule produced :  [3] V(1) & (a.b) -> a
New rule produced :  [4] a+S(b) -> S(a+b)
New rule produced :  [5] V(1) & #(a) -> V(1)
New rule produced :  [6] lambda(a) & b -> lambda(a & #(b))
New rule produced :  [7] V(a) & phi(id,b) -> V(a+b)
New rule produced :  [8] (a app b) & s -> (a & s) app (b & s)
New rule produced :  [9] V(S(a)) & (b.s) -> V(a) & s
New rule produced :  [10] V(1) & phi(#(a),b) -> V(S(b))
New rule produced :  [11] V(S(a)) & #(b) -> V(a) & phi(b,1)
New rule produced :  [12] V(1) & phi(a.b,s) -> a & phi(id,s)
New rule produced :  [13] V(S(a)) & phi(b.s,n) -> V(a) & phi(s,n)
New rule produced :  [14] V(a) & phi(phi(b,s),n) -> V(a) & phi(b,s+n)
New rule produced :  [15] V(S(a)) & phi(#(b),s) -> V(a) & phi(b,S(s))

Result:
{ [1] a+1 -> S(a),
  [2] V(a) & id -> V(a),
  [3] V(1) & (a.b) -> a,
  [4] a+S(b) -> S(a+b),
  [5] V(1) & #(a) -> V(1),
  [6] lambda(a) & b -> lambda(a & #(b)),
  [7] V(a) & phi(id,b) -> V(a+b),
  [8] (a app b) & s -> (a & s) app (b & s),
  [9] V(S(a)) & (b.s) -> V(a) & s,
  [10] V(1) & phi(#(a),b) -> V(S(b)),
  [11] V(S(a)) & #(b) -> V(a) & phi(b,1),
  [12] V(1) & phi(a.b,s) -> a & phi(id,s),
  [13] V(S(a)) & phi(b.s,n) -> V(a) & phi(s,n),
  [14] V(a) & phi(phi(b,s),n) -> V(a) & phi(b,s+n),
  [15] V(S(a)) & phi(#(b),s) -> V(a) & phi(b,S(s)) } (15 rules)

Number of calls to AC matching      : 1075
```

```
Number of successful calls         : 0 (0%)
Number of calls to unification     : 94
Number of unifiers generated       : 15 (0.16 average)
Number of critical pairs considered : 15 (100%)
Number of rules produced           : 15
Number of rules retained           : 15

      times       |   user   | system
------------------+----------+--------
Total times       |    4.550 |    0.517
Unification times |    0.400 |    0.017
Matching times    |    0.917 |    0.100
```

Cqfd. Remarquons qu'ORME est incapable d'orienter la règle (14).

4.4.2 Terminaison de $\lambda\psi$

Interprétations. On va là aussi utiliser une composition d'ordres lexicographiques et polynomiaux. Les interprétations sont très proches, on ajoute celles de $/$ et Ψ :

$$
\begin{array}{lcllcl}
[\![V(n)]\!]_1 & = & [\![n]\!] & [\![V(n)]\!]_2 & = & [\![n]\!] \\
[\![S(n)]\!]_1 & = & [\![n]\!] & [\![S(n)]\!]_2 & = & [\![n]\!]+1 \\
[\![1]\!]_1 & = & 2 & [\![1]\!]_2 & = & 2 \\
[\![(a\ b)]\!]_1 & = & [\![a]\!]+[\![b]\!] & [\![(a\ b)]\!]_2 & = & [\![a]\!]+[\![b]\!]+1 \\
[\![\lambda a]\!]_1 & = & [\![a]\!]+1 & [\![\lambda a]\!]_2 & = & [\![a]\!] \\
[\![a[\sigma]]\!]_1 & = & [\![a]\!][\![\sigma]\!] & [\![a[\sigma]]\!]_2 & = & [\![a]\!]([\![\sigma]\!]+1) \\
[\![a\cdot\sigma]\!]_1 & = & [\![a]\!]+[\![\sigma]\!] & [\![a\cdot\sigma]\!]_2 & = & [\![a]\!]+[\![\sigma]\!] \\
[\![id]\!]_1 & = & 2 & [\![id]\!]_2 & = & 2 \\
[\![\Uparrow(\sigma)]\!]_1 & = & [\![\sigma]\!] & [\![\Uparrow(\sigma)]\!]_2 & = & 4[\![\sigma]\!] \\
[\![\Phi(\sigma,p)]\!]_1 & = & [\![\sigma]\!]([\![p]\!]-1) & [\![\Phi(\sigma,p)]\!]_2 & = & [\![\sigma]\!]([\![p]\!]+1) \\
[\![p+q]\!]_1 & = & [\![p]\!]+[\![q]\!]-2 & [\![p+q]\!]_2 & = & [\![p]\!]+2[\![q]\!] \\
[\![\Psi(p)]\!]_1 & = & [\![p]\!] & [\![\Psi(p)]\!]_2 & = & [\![p]\!] \\
[\![a/]\!]_1 & = & [\![a]\!]+1 & [\![a/]\!]_2 & = & [\![a]\!]
\end{array}
$$

Arguments. Le fichier d'argument est tout-à-fait comparable au précédent :

fichier d'arguments pour $\lambda\psi$

```
%
% System lambda-psi of Rios [Contributions a l'etude des
% lambda-calculs avec des substitutions explicites, These de doctorat,
% Univ. Paris VII, 1993
%
```

```
%
% termination using polynomial interpretations
%
% Xavier Urbain
%
% $Id: lambda_psi.cim,v 1.1 1996/07/16 12:00:34 urbain Exp $

operators
% # = lift, a&b = a[b]
  V,S,lambda,#  : unary
  psi,/        : unary

  1             : constant

  app,&,+       : infix binary
  phi           : binary

  a,b,s,n,p,q,m : variable

axioms

%Beta
%     lambda(a) app b = a&/(b) ;

%App
          (a app b)&s = (a&s) app (b&s) ;
%Lambda
         lambda(a)&s  = lambda(a&#(s)) ;

%FVar
            V(1)&/(a) = a ;
%Rvar
         V(S(n))&/(a) = V(n) ;

%FVarLift
            V(1)&#(s) = V(1) ;
%RVarLift
         V(S(n))&#(s) = V(n)&phi(s,1) ;

%FVarPhiSI
       V(1)&phi(/(a),p) = a&psi(p) ;
%RVarPhiSI
    V(S(n))&phi(/(a),p) = V(S(n+p)) ;
```

```
%FVarPhiLift
     V(1)&phi(#(s),p) = V(S(p)) ;
%RVarPhiLift
  V(S(n))&phi(#(s),p) = V(n)&phi(s,S(p)) ;


%VarPhiPhi
 V(n)&phi(phi(s,q),p) = V(n)&phi(s,(q+p)) ;
%VarPhiPsi
   V(n)&phi(psi(q),p) = V((n+q)+p) ;

%VarPsi
          V(n)&psi(p) = V(n+p) ;

%Plus1
                  n+1 = S(n) ;
%PlusS
                n+S(m) = S(n+m) ;


order

  lexico(
    poly([V](n)=n; [S](n)=n; [1]=2; [app](a,b)=a+b; [lambda](a)=a+1;
[&](a,s)=a.s; [#](s)=s; [phi](s,p)=s.(p-1); [+](p,q)=p+q-2;
[psi](p)=p; [/](a)=a+1) ;
    poly([V](n)=n; [S](n)=n+1; [1]=2; [app](a,b)=a+b+1; [lambda](a)=a;
[&](a,s)=a.(s+1); [#](s)=4.s; [phi](s,p)=s.(p+1); [+](p,q)=p+2.q;
[psi](p)=p; [/](a)=a))

end
```

Résultat. C*i*ME oriente les règles de la façon suivante :

```
> cime cime/examples/explicit-subsitutions/lambda_psi
Welcome to CiME v1.11 - SunOS-5.4 - 07/18/96 16:50:22


New rule produced :  [1] a+1 -> S(a)
New rule produced :  [2] V(1) & /(a) -> a
New rule produced :  [3] a+S(b) -> S(a+b)
New rule produced :  [4] V(1) & #(a) -> V(1)
New rule produced :  [5] V(S(a)) & /(b) -> V(a)
New rule produced :  [6] lambda(a) & b -> lambda(a & #(b))
```

```
New rule produced :  [7] V(a) & psi(b) -> V(a+b)
New rule produced :  [8] (a app b) & s -> (a & s) app (b & s)
New rule produced :  [9] V(1) & phi(#(a),b) -> V(S(b))
New rule produced :  [10] V(1) & phi(/(a),b) -> a & psi(b)
New rule produced :  [11] V(S(a)) & #(b) -> V(a) & phi(b,1)
New rule produced :  [12] V(a) & phi(psi(b),s) -> V((a+b)+s)
New rule produced :  [13] V(S(a)) & phi(/(b),s) -> V(S(a+s))
New rule produced :  [14] V(a) & phi(phi(b,s),n) -> V(a) & phi(b,s+n)
New rule produced :  [15] V(S(a)) & phi(#(b),s) -> V(a) & phi(b,S(s))

Result:
{ [1] a+1 -> S(a),
  [2] V(1) & /(a) -> a,
  [3] a+S(b) -> S(a+b),
  [4] V(1) & #(a) -> V(1),
  [5] V(S(a)) & /(b) -> V(a),
  [6] lambda(a) & b -> lambda(a & #(b)),
  [7] V(a) & psi(b) -> V(a+b),
  [8] (a app b) & s -> (a & s) app (b & s),
  [9] V(1) & phi(#(a),b) -> V(S(b)),
  [10] V(1) & phi(/(a),b) -> a & psi(b),
  [11] V(S(a)) & #(b) -> V(a) & phi(b,1),
  [12] V(a) & phi(psi(b),s) -> V((a+b)+s),
  [13] V(S(a)) & phi(/(b),s) -> V(S(a+s)),
  [14] V(a) & phi(phi(b,s),n) -> V(a) & phi(b,s+n),
  [15] V(S(a)) & phi(#(b),s) -> V(a) & phi(b,S(s)) } (15 rules)

Number of calls to AC matching      : 958
Number of successful calls          : 0 (0%)
Number of calls to unification      : 94
Number of unifiers generated        : 15 (0.16 average)
Number of critical pairs considered : 15 (100%)
Number of rules produced            : 15
Number of rules retained            : 15

      times       |   user   | system
------------------+----------+--------
Total times       |    4.583 |    0.517
Unification times |    0.467 |    0.000
Matching times    |    0.917 |    0.183
```

Cqfd. La règle (14) n'ayant pas changé, elle n'est toujours pas orientée par ORME.

Chapitre 5

Puisqu'il faut conclure...

Nous avons réalisé une implantation des ordres polynomiaux dans le système CiME, propre et modulaire, en privilégiant l'abstraction des objets manipulés. Les exemples concernant l'orientation de règles sont traités avec succès. L'insertion dans le système a été faite avec le minimum de bouleversements. CiME peut enfin orienter des systèmes nécessitant des interprétations polynomiales.

Mais la capacité à ordonner les polynômes ne peut tout remplacer. En effet, Dershowitz a exhibé certains systèmes où les interprétations polynomiales échouent et il existe des formes d'interprétations par des fonctions non-polynomiales, par exemples exponentielles. Si CiME est efficace sur certains λ-calculs avec substitutions explicites (et même plus performant qu'ORME sur $\lambda\phi$ et $\lambda\psi$) il lui manque encore des interprétations élémentaires : il est gênant de ne pouvoir prouver la terminaison d'un calcul aussi intéressant que $\lambda\upsilon$ à cause d'une incapacité à gérer les interprétations exponentielles ! CiME mériterait de voir sa puissance et sa polyvalence accrues par une implantation de ce type, gageons que ce sera bientôt chose faite.

Bibliographie

[1] M. Abadi, L. Cardelli, P.-L. Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 31–46, San Francisco, 1990.

[2] Ahlem Ben Cherifa and Pierre Lescanne. An actual implementation of a procedure that mechanically proves termination of rewriting systems based on inequalities between polynomial interpretations. In *Proc. 8th Int. Conf. on Automated Deduction, Oxford, England, LNCS 230*, pages 42–51. Springer-Verlag, July 1986.

[3] Evelyne Contejean and Claude Marché. CiME: Completion Modulo E. In Harald Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 416–419, New Brunswick, NJ, USA, July 1996. Springer-Verlag. System Description.

[4] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[5] M. Davis and M. Hersh. Hilbert's 10th problem. *Scientific American*, 229(5):84–91, 1973.

[6] M. Davis, H. Putnam, and J. Robinson. The decision problem for exponential Diophantine equations. *Annals of Mathematics*, 74:425–436, 1961.

[7] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5):380–392, 1972.

[8] N. G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. *Proc. of the Koninklijke Nederlands Akademie*, 81(3):1–9, September 1978. Dedicated to A. Heyting on the occasion of his 80th birthday on May 9,1978.

[9] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, February 1987.

[10] Jean-Pierre Jouannaud. Introduction to rewriting. In Hubert Comon and Jean-Pierre Jouannaud, editors, *Term Rewriting*, volume 909 of *Lecture Notes in Computer Science*. French Spring School of Theoretical Computer Science, Springer-Verlag, 1995.

[11] Dallas S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979.

[12] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions. In *Conference Record of the 21st Symposium on Principles of Programming Languages*, Portland, Oregon, 1994. ACM.

[13] Pierre Lescanne and Jocelyne Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In Jieh Hsiang, editor, *6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, Kaiserslautern, Germany, April 1995. Springer-Verlag.

[14] Claude Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité des problèmes du mot dans certains classes de théories équationnelles.* Thèse de doctorat, Université Paris-Sud, Orsay, France, October 1993.

[15] A. Ríos. *Contributions à l'étude des λ-calculs avec des substitutions explicites.* Thèse de doctorat, Université Paris VII, 1993.