Centre de Recherche en Informatique de Nancy - INRIA Lorraine - Équipe EURÉCA

Mémoire de DEA

Université Henri Poincaré NANCY I
UFR STMIA
Département Informatique

Une machine à environnement avec partage
La UP-machine

présenté le 6 Septembre 1994
par Frédéric Lang

Composition du jury :

Didier Galmiche
Pierre Lescanne
Jocelyne Rouyer
Claude Kirchner
Philippe de Groote

Table des matières

Table des figures

Chapitre 1

Introduction

Ce document est une présentation du travail que j'ai effectué dans le cadre de mon stage de recherche pour le Diplôme d'Études Approfondies (DEA) en Informatique, formation suivie à l'université Henri Poincaré NANCY I. Le travail présenté a été réalisé au sein de l'équipe de recherche EURÉCA du laboratoire d'informatique *CRIN - INRIA Lorraine*, sous la direction de Jocelyne Rouyer et Pierre Lescanne.

Le λ-calcul est le formalisme de base de la programmation fonctionnelle et des systèmes de preuve de programme. Une partie des recherches sur le λ-calcul se porte actuellement vers des techniques de réduction efficaces pour l'implantation des langages et des systèmes de preuve, parmi lesquelles certaines se tournent vers les systèmes de réécriture ; c'est la raison de l'apparition des λ-calcul avec substitutions explicites, qui permettent d'utiliser la réécriture pour l'évaluation, et notamment du système $\lambda\upsilon$. Ces systèmes peuvent-être implantés sous la forme de machines à environnements, ou de machines à réducteurs de graphes ; la U-machine est une machine à environnements qui implante le système $\lambda\upsilon$. Afin d'optimiser cette machine, on cherche à y implanter le partage de sous-termes, c'est-à-dire qu'on cherche à limiter le nombre de réécritures effectuées en «factorisant» les réécritures communes à plusieurs évaluations d'un même argument dans un terme. La machine obtenue se nomme UP-machine.

Dans un premier temps, je fais un rappel rapide sur le λ-calcul et sa syntaxe, puis je présente le $\lambda\upsilon$-calcul de Pierre Lescanne, principale base théorique de mon travail. Je résume ensuite le travail de Pierre Crégut sur la machine de Krivine, et montre comment il a implanté le partage sur cette machine. Puis je présente la U-machine que je modifierai de la même façon que Pierre Crégut, pour obtenir la UP-machine. Je termine enfin par quelques résultats de la machine réalisée, qui seront comparés à ceux de la U-machine.

Chapitre 2

Le λ-calcul non typé

Ce chapitre fait un rappel très rapide du λ-calcul afin de fixer la syntaxe utilisée.

2.1 Origines et motivations du λ-calcul

Le λ-calcul est un formalisme simple et expressif introduit par Alonzo Church dans les années 1930, qui avait un but initial de fondation des mathématiques [Chu32a, Chu32b]. Ce but a été rapidement abandonné[1], et le λ-calcul ne connut un renouveau qu'avec le développement de l'informatique ; il est notamment à la base de la programmation fonctionnelle, et est utilisé comme outil pour la déduction de preuve automatique. De nombreux enrichissements du λ-calcul existent ; le λ-calcul que je présente ici est le plus primitif : le λ-calcul non typé ou λ-calcul simple.

2.2 Syntaxe du λ-calcul non typé

Les définitions données sont tirées de [Bar84, Lal90], mais la syntaxe introduite en est un peu différente pour des questions d'habitudes et de conventions utilisées dans les travaux sur lesquels mon travail s'est appuyé.

2.2.1 Formation des termes

On introduit les symboles 'λ','(', et ')', ainsi qu'un ensemble infini dénombrable de *variables* noté X. Les règles de formation des *termes* du λ-calcul, ou λ-*termes*, sont données par la définition 1.

1. Il revient au goût du jour, notamment à la suite des travaux de Martin-Løf

Définition 1

i) *toute variable est un terme*

ii) *si x est une variable et M un terme, alors $\lambda x.M$ est un terme*

iii) *si M et N sont des termes, alors $(M\ N)$ est un terme*

EXEMPLE x, $\lambda x.(x\ x)$, $\lambda x.\lambda y.(x\ y)$, $(\lambda x.x\ y)$ sont des termes du λ-calcul.

On définit la notion d'occurrence d'un terme dans un autre terme :

Définition 2

i) *X a une occurrence dans X*

ii) *si X a une occurrence dans U ou dans V, alors X a une occurrence dans $(U\ V)$*

iii) *si X a une occurrence dans U, alors pour tout y, X a une occurrence dans $\lambda y.U$.*

L'interprétation exacte des λ-termes varie dans les différentes applications de la théorie. En général, chaque λ-terme représente une fonction, dont les valeurs ou les arguments peuvent eux-mêmes être des fonctions. Les variables représentent des fonctions quelconques, et $(X\ Y)$ représente l'application de la fonction X à l'argument Y. Si Y est un terme dans lequel x a des occurrences libres (voir la définition plus loin), alors $\lambda x.Y$ représente la fonction dont la valeur en un argument A est le résultat de la substitution de A pour x dans Y (voir la section 2.4 sur la β-réduction).

EXEMPLE Dans un λ-calcul enrichi avec l'opérateur $+$, le terme $(+\ x\ 1)$ représente l'application de la fonction $+$ à x et à 1, et $\lambda x.(+\ x\ 1)$ représente la fonction $x \mapsto x + 1$.

Pour les applications, on notera plus simplement $(x_1\ \ldots\ x_n)$ le terme $(\ldots((x_1\ x_2)\ x_3)\ \ldots\ x_n)$.

On peut représenter les λ-termes sous forme d'arbres : les feuilles sont les variables, les nœuds abstraction ont pour fils gauche une variable et pour fils droit un arbre (fig. 2.1), les nœuds application ont pour fils gauche et droit des arbres (fig. 2.2).

EXEMPLE Le terme $\lambda x.(\lambda y.(x\ y)\ x)$ a la représentation arborescente de la figure 2.3.

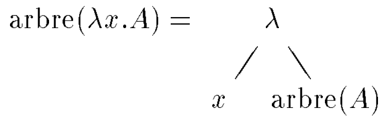

FIG. 2.1 – *Représentation arborescente d'un nœud abstraction $\lambda x.A$*

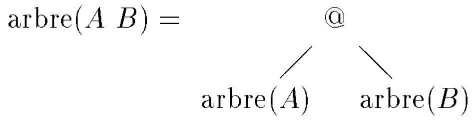

FIG. 2.2 – *Représentation arborescente d'un nœud application $(A\ B)$*

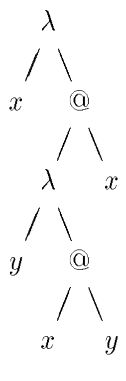

FIG. 2.3 – *Représentation arborescente du terme $\lambda x.(\lambda y.(x\ y)\ x)$*

```
                 @
                / \
               λ   x libre
              / \
déclaration x    @
                / \
          liée x   y libre
```

FIG. 2.4 – *Variables libres et liées dans le terme* $(\lambda x.(x\ y)\ x)$.

2.2.2 Variables libres et variables liées

Dans les termes du λ-calcul, on différencie les variables dites *libres* des variables dites *liées*.

Définition 3 *Une occurrence de variable x dans un terme Y est liée si et seulement si elle est dans une partie de Y de la forme $\lambda x.Z$; sinon, elle est libre. On dit que x est libre dans Y quand x a une occurrence libre dans Y.*

> EXEMPLE Dans le terme $(\lambda x.(x\ y)\ x)$ la première occurrence de x est liée par λx, et la deuxième est libre. La variable y est libre (voir fig. 2.4).

2.2.3 Notation de De Bruijn

De Bruijn a proposé en 1972 une forme pour la représentation des variables, très utile pour l'implantation, qui utilise le niveau d'abstraction des variables liées [DB72]. Le niveau d'abstraction d'une occurrence de variable liée est le nombre de nœuds λ qui la séparent du nœud λ qui la déclare, lorsqu'on remonte dans l'arbre du terme.

La notation de De Bruijn des termes du λ-calcul est obtenue en remplaçant toutes les variables liées par leur niveau d'abstraction. Cette notation a pour avantage de ne pas nécessiter de renommage[2] pour comparer deux termes clos (i.e dont toutes les variables sont liées) identiques.

> EXEMPLE Les termes $\lambda x.x$ et $\lambda y.y$ sont identiques au nom de leur variable près. Tous deux ont la même représentation en notation de De Bruijn[3] : $\lambda\,\underline{1}$.

Ainsi tout terme clos a une représentation unique en notation de De Bruijn.

Pour la représentation en arbre, les nœuds abstraction deviennent des nœuds unaires puisqu'on ne déclare plus de nom (fig. 2.5).

2. On appelle renommage ou α-conversion l'opération qui consiste à changer le nom des variables afin d'éviter les collisions.

3. Suivant la convention que l'on utilise, on peut également trouver $\lambda\,\underline{0}$.

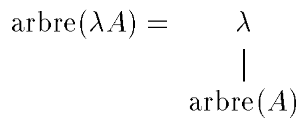

Fig. 2.5 – *Représentation arborescente d'un nœud abstraction λA en notation de De Bruijn*

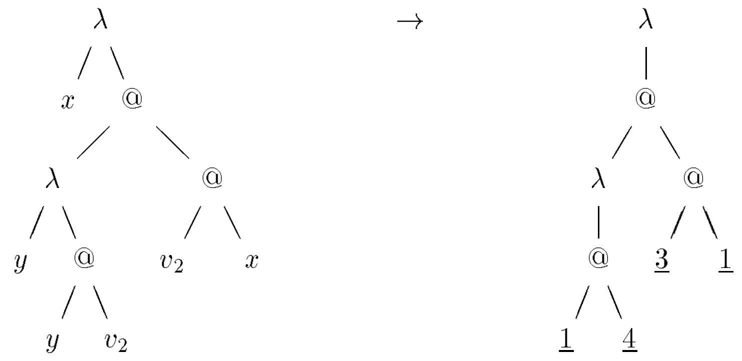

Fig. 2.6 – *Notation de De Bruijn du terme $\lambda x.(\lambda y.(y\ v_2)\ (v_2\ x))$*

Pour les variables libres, qu'il faut également représenter par des entiers pour respecter la cohérence de la notation, on utilise les entiers supérieurs au niveau maximal d'abstraction : à chaque nom de variable libre est associé un entier, auquel on ajoute le nombre d'abstractions qui séparent l'occurrence considérée de la racine.

> EXEMPLE Soit $X_l = \{v_1, v_2, \ldots\}$ l'ensemble des variables libres : les variables libres sont représentées par leur indice augmenté du niveau d'abstraction maximal. Le terme $\lambda x.(\lambda y.(y\ v_2)\ (v_2\ x))$ est représenté par $\lambda(\lambda(\underline{1}\ \underline{4})\ (\underline{3}\ \underline{1}))$ (voir la figure 2.6).

Cependant, la plupart du temps, on ne manipule que des termes clos (i.e. dont toutes les variables sont liées) et on n'est pas confronté au problème de représentation des variables libres.

Bien entendu, l'inconvénient de la notation de De Bruijn est qu'elle est peu lisible, c'est pourquoi elle n'est utilisée que pour l'implantation.

2.3 Substitution

On appelle *substitution* d'un terme à une variable le remplacement de toutes les occurrences libres de cette variable par le terme considéré. En notation de

$$\begin{array}{rcll}
x[N/x] & \equiv & N & \\
a[N/x] & \equiv & a & \text{où } a \text{ est une variable et } a \not\equiv x \\
(M_1\ M_2)[N/x] & \equiv & (M_1[N/x])\ (M_2[N/x]) & \\
(\lambda x.Y)[N/x] & \equiv & \lambda x.Y & \\
(\lambda y.Y)[N/x] & \equiv & \lambda y.(Y[N/x]) & \text{si } y \notin N \text{ ou } x \notin Y\ \ (y \not\equiv x) \\
 & \equiv & \lambda z.(Y[z/y][N/x]) & \text{si } y \in N \text{ et } x \in Y
\end{array}$$

FIG. 2.7 – *Définition formelle de la substitution*

De Bruijn, ceci implique une mise à jour des variables libres du terme substitut en fonction de leur nouvel indice d'abstraction dans le terme résultat [4]. Avec la notation classique, ceci implique un renommage des variables liées du terme initial qui pourraient lier des variables libres du terme substitut. On note $M[N/x]$ la substitution du terme N à la variable x

> EXEMPLE Le terme $\lambda x.(x\ y)[\lambda x.x/y]$ est équivalent au terme $\lambda z.(z\ \lambda x.x)$ où z est le nouveau nom donné à la variable x du terme initial $\lambda x.(x\ y)$. En notation de De Bruijn, aucun renommage n'est nécessaire, le terme substitut ne contenant pas de variables libres.

Plus formellement, le mécanisme de substitution peut être décrit par le système de règles de la figure 2.7.

Ce mécanisme de substitution est essentiel en programmation fonctionnelle pour implanter le passage de paramètres lors de l'appel d'une fonction, qui est décrit par une relation entre termes, la β-réduction.

2.4 β-réduction

Un terme de la forme $(\lambda x.M\ N)$ est appelé un radical : c'est la rencontre d'une abstraction et d'une application. Le mécanisme essentiel du λ-calcul (et de la programmation fonctionnelle) est la contraction des radicaux. La transformation du radical $(\lambda x.M\ N)$ en $M[N/x]$ s'appelle une β-contraction. Cette transformation exprime le passage d'un argument à une fonction.

La β-réduction est fondée sur la contraction des radicaux. Elle est décrite par un système d'inférence comportant la règle de contraction et trois autres règles (voir figure 2.8). En plus clair, ces règles expriment la contraction des radicaux, le prolongement de la contraction sous les abstractions, et sous les deux membres des applications.

On notera $M \xrightarrow[\beta]{} N$ s'il existe une dérivation fermée (i.e. sans hypothèse) du jugement $M \triangleright N$ par le système d'inférence de la figure 2.8, et on dira que M

4. j'appelle terme substitut le terme placé entre les crochets, et terme résultat le terme obtenu en résultat de la substitution.

$$\frac{}{(\lambda x.P\ Q) \rhd P[Q/x]} \quad (RAD)$$

$$\frac{M \rhd M'}{\lambda x.M \rhd \lambda x.M'} \quad (\lambda)$$

$$\frac{M \rhd M'}{(M\ N) \rhd (M'\ N)} \quad (1)$$

$$\frac{M \rhd M'}{(N\ M) \rhd (N\ M')} \quad (2)$$

FIG. 2.8 – *Définition formelle de la β-réduction*

se réduit immédiatement en N. La β-réduction est la relation $\xrightarrow[\beta]{*}$, fermeture transitive de la relation $\xrightarrow[\beta]{}$.

2.5 Propriétés fondamentales du λ-calcul

Je donne dans cette section quelques définitions et propriétés importantes du λ-calcul.

Définition 4 *On dit qu'un terme est en β-forme normale forte s'il ne contient pas de radical.*

> EXEMPLE Les termes x, $\lambda x.(x\ y)$, $(x\ \lambda x.x)$ sont en β-forme normale forte. Le terme $\lambda x.(\lambda y.x\ z)$ n'est pas en β-forme normale forte, car il contient un radical.

Définition 5 *On dit qu'un terme est en β-forme normale faible, ou β-forme normale de tête, lorsque le terme ne contient plus de radical en tête, c'est-à-dire lorsqu'il n'est pas de la forme $\lambda x.A\ B_1 \ldots B_n$ $(n \geq 1)$*

Par la suite, on emploiera β-forme normale, ou plus simplement forme normale pour signifier β-forme normale forte.

Définition 6 *On dit qu'un terme M est normalisable s'il existe une β-réduction $M \xrightarrow[\beta]{*} N$ où N est en forme normale.*

Un des problèmes du λ-calcul est que tous les termes ne sont pas normalisables, notamment que l'évaluation de certains termes peut être infinie (par exemple la réduction du terme $(\lambda x.(x\ x)\ \lambda x.(x\ x))$ qui se réécrit sur lui même).

D'autre part, le problème de la normalisation n'est pas décidable (résultat dû à Church), c'est-à-dire que l'on ne peut pas affirmer qu'un terme n'est pas normalisable sans tenter de le normaliser. Remarquons qu'on peut réduire un terme suivant plusieurs stratégies (réduction du radical le plus interne, le plus externe, etc.) qui ne mènent pas toutes au même résultat : si la réduction d'un terme peut diverger avec une stratégie de calcul, elle ne diverge pas forcément avec toutes les stratégies.

Définition 7 *Un terme est dit fortement normalisable s'il n'existe pas de réduction infinie* $M \xrightarrow[\beta]{} M_1 \xrightarrow[\beta]{} M_2 \xrightarrow[\beta]{} \dots$

On doit donc choisir une stratégie de réduction afin de rendre le λ-calcul opératoire. Par exemple la stratégie de réduction à gauche (i.e. on commence par réduire la partie gauche des applications avant la partie droite) qui est dite normalisante (i.e. la réduction qu'elle détermine pour tout terme normalisable est finie. On dit aussi que la stratégie est sûre).

Une autre propriété importante du λ-calcul est la confluence.

Définition 8 *On dit que la relation binaire* $\longrightarrow$ *sur* E *est confluente en* x *si pour tout* x_1, x_2*, si* $x \xrightarrow{*} x_1$ *et* $x \xrightarrow{*} x_2$*, alors il existe* x' *tel que* $x_1 \xrightarrow{*} x'$ *et* $x_2 \xrightarrow{*} x'$*, et qu'elle est confluente sur* E *si elle l'est en tout point de* E*.*

La confluence est une propriété fondamentale de toute modélisation du calcul comme processus de réduction : elle signifie l'indépendance du résultat du calcul vis-à-vis du chemin de réduction suivi.

Chapitre 3

Le $\lambda\upsilon$-calcul

3.1 Les λ-calculs avec substitutions explicites

Les λ-calculs avec substitutions explicites ont été introduits récemment. Il en existe quatre principaux : le $\lambda\sigma$-calcul proposé par Abadi, Cardelli, Curien et Lévy [ACCL90], le $\lambda\sigma_{\Uparrow}$-calcul proposé par Hardin et Lévy [HL89, CHL92], le $\lambda\tau$-calcul [Río93], et enfin le $\lambda\upsilon$-calcul (lire *lambda upsilon*) proposé par Pierre Lescanne [Les94, LR93]. La différence entre le λ-calcul classique et les calculs avec substitutions explicites est que le mécanisme de substitution, habituellement décrit à un niveau supérieur par un formalisme spécifique et externe, contient dans le même système la règle β et une description de l'évaluation de la substitution. Les λ-calculs avec substitutions explicites sont des systèmes de réécriture de termes du premier ordre qui permettent des implantations du λ-calcul élégantes et uniformes. Ce sont de bons candidats pour des implantations efficaces dans les langages de programmation fonctionnelle, ou comme machines abstraites.

3.2 Le $\lambda\upsilon$-calcul

Cette section fait un bref résumé de ce qu'est le $\lambda\upsilon$-calcul. Pour plus de détails, il est conseillé de se référer à [Les94, LR93], notamment pour la preuve de correction.

Le $\lambda\upsilon$-calcul est un calcul avec substitutions explicites qui contient un ensemble minimal d'opérateurs, c'est à dire uniquement les opérateurs strictement nécessaires pour décrire le calcul ; pour cela, il se veut simple.

Il y a quatre opérateurs sur les termes, *abstraction* ($\lambda\_$), *application* ($\_\ \_$), *fermeture* $\_[\_]$ et *variables* ($\underline{n}$), et trois opérateurs sur les substitutions, *slash* ($\_/$), *lift* ($\Uparrow(\_)$) et *shift* ($\uparrow$). L'opérateur *fermeture* introduit les substitutions dans le calcul. $\lambda\upsilon$ utilise la notation de De Bruijn pour les variables qui sont notées $\underline{1}, \underline{2}, \ldots, \underline{n}, \underline{n+1}, \ldots$ Un terme qui ne contient pas de fermetures est appelé un terme pur.

Les termes du $\lambda\upsilon$-calcul sont définis par la grammaire de la figure 3.1, et le système de la figure 3.2 décrit le $\lambda\upsilon$-calcul.

Termes	a	$::=$	$\underline{n} \mid a\ b \mid \lambda a \mid a\,[s]$
Substitutions	s	$::=$	$a/ \mid\ \Uparrow(s) \mid\ \uparrow$
Variables	n	$::=$	$\underline{1} \mid \underline{n+1}$

FIG. 3.1 – *Grammaire des termes du $\lambda\upsilon$-calcul*

(Beta)	$(\lambda a)\ b$	$\rightarrow$	$a\,[b/]$
(App)	$(a\ b)\,[s]$	$\rightarrow$	$a\,[s]\,b\,[s]$
(Lambda)	$(\lambda a)\,[s]$	$\rightarrow$	$\lambda(\,a\,[\Uparrow(s)])$
(Fvar)	$\underline{1}\,[a/]$	$\rightarrow$	a
(Rvar)	$\underline{n+1}\,[a/]$	$\rightarrow$	$\underline{n}$
(Fvarlift)	$\underline{1}\,[\Uparrow(s)]$	$\rightarrow$	$\underline{1}$
(Rvarlift)	$\underline{n+1}\,[\Uparrow(s)]$	$\rightarrow$	$\underline{n}\,[s][\uparrow]$
(Varshift)	$\underline{n}\,[\uparrow]$	$\rightarrow$	$\underline{n+1}$

FIG. 3.2 – *Le système $\lambda\upsilon$*

La règle (Beta) est la règle de β-réduction classique (voir chapitre 2 section 2.4). Elle introduit la fermeture $[b/]$ dans le calcul qui est intuitivement équivalente à la substitution $[b/x]$ de la notation classique. Ici la variable n'est pas explicitement nommée, car c'est sur la variable déclarée par le λ réduit, dont les indices sont connus par calcul ($\underline{1}$, puis $\underline{n+1}$ à chaque fois qu'on passera sous un λ), que doit s'opérer la substitution du terme b. On note que les indices des variables du terme b n'ont pas lieu de changer tant que b reste au même niveau d'abstraction. Par contre, au passage sous les λ (règle (Lambda)) il faut recalculer les indices des variables libres du terme substitut, puisqu'on le place à un niveau d'abstraction supplémentaire. C'est la raison de l'introduction de l'opérateur $\Uparrow$, et indirectement de l'opérateur $\uparrow$, introduit par $\Uparrow$ (règle Rvarlift).

La suppression du λ d'un radical réduit doit donner lieu à une mise-à-jour des indices des variables libres et des variables liées à un λ situé plus haut dans le terme[1] (règle (Rvar)).

La règle (App) permet de transmettre les fermetures aux deux membres des applications.

la règle (Fvar) réalise explicitement la substitution de la variable par le terme.

(Fvarlift), (Rvarlift) et (Varshift) sont les trois règles qui définissent le calcul des opérateurs *lift* ($\Uparrow$) et *shift* ($\uparrow$).

Intuitivement $b/, \Uparrow(s)$ et $\uparrow$ ont la signification suivante :

$$
\begin{array}{lcrcl}
b/ & : & \underline{1} & \mapsto & b \\
 & & \underline{2} & \mapsto & \underline{1} \\
 & & & \vdots & \\
 & & \underline{n+1} & \mapsto & \underline{n} \\
 & & & \vdots & \\
 & & & & \\
\Uparrow(s) & : & \underline{1} & \mapsto & \underline{1} \\
 & & \underline{2} & \mapsto & s(\underline{1})\,[\uparrow] \\
 & & & \vdots & \\
 & & \underline{n} & \mapsto & s(\underline{n})\,[\uparrow] \\
 & & & \vdots & \\
 & & & & \\
\uparrow & : & \underline{1} & \mapsto & \underline{2} \\
 & & \underline{2} & \mapsto & \underline{3} \\
 & & & \vdots & \\
 & & \underline{n} & \mapsto & \underline{n+1} \\
 & & & \vdots &
\end{array}
$$

Ainsi, le mécanisme de substitution est décrit au même niveau que le mécanisme de réduction. Ce système est donc un système de réécriture, simple à mettre en œuvre.

1. On dit qu'un λ est situé plus haut dans le terme si on trouve ce λ sur le chemin menant du nœud courant à la racine de l'arbre qui représente ce terme. Il est pertinent de remarquer qu'une telle variable peut être considérée comme une variable libre du radical considéré.

Chapitre 4

Les machines K et KP

Ce chapitre décrit un travail qui a été effectué par Pierre Crégut dans le cadre de sa thèse de doctorat et qui est détaillé dans [Cré91]. J'ai largement réutilisé les idées développées dans ce document pour réaliser mon travail.

4.1 Machines abstraites

Certains compilateurs des langages de programmation impératifs sont composés d'une partie frontale - la partie d'analyse, qui génère un code intermédiaire - et d'une partie finale - la partie de génération de code. Ce découpage en deux parties du compilateur facilite sa modification lorsqu'on souhaite qu'il produise du code pour une autre machine. La machine abstraite d'un langage de programmation impératif est une des formes de représentation possibles du code intermédiaire [ASU91].

Les machines abstraites des langages impératifs sont également appelées machines à pile, car elles utilisent une structure de pile pour évaluer les instructions. Elles sont constituées également d'une zone de code (instructions et compteur ordinal), et d'une zone de données.

En programmation fonctionnelle, on peut distinguer deux grandes classes de machines abstraites [Cré91] :

- les **machines à environnement**, qui sont l'aboutissement des techniques de compilation des langages procéduraux (e.g. Pascal), et qui distinguent les fonctions vues comme opérateurs (représentées par du code machine) des fonctions vues comme arguments (stockées dans une structure de données appelée *environnement*) ;

- les **machines pour réducteurs de graphes** qui sont fidèles à la tradition des systèmes de réécriture en refusant la distinction code-données.

Nous nous intéresserons à la première catégorie, dont font partie les machines K, KP, U et UP décrites plus loin.

$$
\begin{aligned}
((C_1\ C_2), e, p) &\rightarrow (C_1, e, <C_2, e>:: p) && (4.1)\\
(\lambda C, e, f :: p) &\rightarrow (C, f :: e, p) && (4.2)\\
(\underline{1}, <C_2, e_2>:: e_1, p) &\rightarrow (C_2, e_2, p) && (4.3)\\
(\underline{n+1}, f :: e, p) &\rightarrow (\underline{n}, e, p) && (4.4)
\end{aligned}
$$

FIG. 4.1 – *La machine de Krivine*

4.2 La machine de Krivine

La machine de Krivine (K), proposée par Jean-Louis Krivine [Kri85], est une machine à environnement, interprète simple du λ-calcul utilisant la notation de De Bruijn pour l'accès aux variables, et la stratégie d'appel par nécessité (i.e. la stratégie d'évaluation de l'opérateur - membre gauche de l'application - avant l'argument - membre droit). Cette machine ne réduit que des termes clos et s'arrête à la forme normale de tête faible (i.e. on ne réduit que les radicaux de tête).

On travaille sur des états constitués d'un code (terme pur du λ-calcul), un environnement, et une pile. L'environnement et la pile sont des listes de fermetures, qui sont des couples <code,environnement>. L'environnement représente les substitutions à opérer suite à des β-contractions, et la pile permet de mémoriser les arguments des applications, ainsi que l'environnement dans lequel ces arguments se trouvaient quand ils ont été empilés[1]. La machine de Krivine est donc décrite par les quatre règles de réécriture de la figure 4.1. Les règles ont la signification intuitive suivante :

- La règle 4.1 permet de stocker l'argument de l'application et l'environnement dans lequel on l'a rencontré sur le sommet de la pile, pour pouvoir continuer l'évaluation de la fonction.

- La règle 4.2 permet de dépiler le dernier argument, qui est le terme à substituer à la dernière variable formelle déclarée (liée au λ de tête du terme situé dans le champ code du membre gauche de la règle), pour le placer dans l'environnement.

- La règle 4.3 permet d'effectuer la substitution de la variable avec le terme situé en tête d'environnement. L'environnement courant devient l'environnement dans lequel se situait l'argument lorsqu'on l'a empilé. Le précédent environnement est abandonné.

- La règle 4.4 permet d'accéder dans l'environnement au terme à substituer à la variable. On remarque que la variable contient le chemin d'accès dans

1. On peut faire l'analogie avec les appels imbriqués de fonctions dans les langages procéduraux : lorsqu'on appelle une fonction, on place ses arguments au sommet de la pile.

l'environnement, i.e. son numéro correspond au nombre de fermetures à remonter dans l'environnement pour retrouver le terme substitut.

La stratégie utilisée pour l'évaluation est une stratégie d'appel par nécessité, c'est à dire qu'on évalue l'argument à chaque fois qu'on en a besoin. De fait, elle est sûre, mais elle n'est pas efficace.

En effet, la règle 4.1 effectue une duplication des environnements, et par conséquent une duplication des fermetures qu'ils contiennent. Ces fermetures n'étant pas évaluées au moment où elles sont dupliquées, il faudra les évaluer autant de fois qu'on voudra les utiliser, i.e. si on utilise un même argument n fois, on l'évalue n fois.

On aimerait pouvoir évaluer l'argument une seule fois, et réutiliser sa forme évaluée. La stratégie en appel par valeur permet cela, puisqu'elle réalise l'évaluation de l'argument avant l'opérateur - et c'est donc la forme évaluée de l'argument qui est dupliquée - mais elle n'est pas sûre, i.e. elle n'est pas normalisante, dans le sens où l'évaluation d'un terme normalisable peut ne pas mener à la forme normale de ce terme. Une troisième stratégie envisageable est l'appel paresseux, c'est à dire qu'on évalue l'argument une fois pour toutes lorsqu'on en a besoin, et qu'on stocke sa forme évaluée. Cette stratégie est implantée grâce au partage des fermetures et donne lieu à la machine KP.

4.3 La machine KP : implantation du partage

Pour implanter le partage, on a besoin de savoir quelle partie de l'évaluation d'un argument est commune à toutes ses évaluations. En effet, on l'a vu, l'évaluation d'un terme se fait dans un certain contexte, composé de l'environnement et de l'état de la pile. Ce contexte est différent pour chacune des utilisations de l'argument. On doit pouvoir reconnaître quelle partie de l'évaluation d'une fermeture est liée à son environnement propre, de celle qui est liée à l'environnement dans lequel elle est plongée.

Pour cela on utilise une marque (notée M), qui permet d'isoler la pile lors de l'évaluation des arguments. Ainsi, l'évaluation des fermetures peut utiliser la partie de la pile située au-dessus de cette marque sans avoir accès aux arguments empilés précédemment, et qui font partie du contexte global dans lequel on évalue l'argument courant. Une fois que cette marque devient gênante pour la poursuite de l'évaluation, elle doit être dépilée. Cela signifie que l'évaluation de l'argument hors contexte est terminée. Pour permettre le partage, les fermetures sont remplacées par leur référence (notée $\sharp a$) dans un état mémoire. Lorsque l'évaluation hors contexte d'un argument est terminée, il faut remplacer dans l'état mémoire la fermeture non évaluée par la fermeture que l'on vient de calculer. C'est pourquoi on conserve dans la marque l'adresse de la fermeture que l'on évalue.

Les fermetures sont stockées dans l'état mémoire, qui est une liste modifiable d'associations [adresse $\leftarrow$ fermeture]. La pile et les environnements sont représentés par des listes d'adresses de fermetures, et la pile peut contenir des marques dans lesquelles sont stockées des adresses. On introduit sur les états mémoire la fonction *nouveau(s)* qui à un état mémoire associe une adresse non

$$
\begin{array}{rcll}
((C_1\ C_2), e, p, s) & \rightarrow & (C_1, e, \sharp a :: p, s[a \leftarrow <C_2, e>) & (4.5)\\
 & & \text{où } a = \text{nouveau}(s) & \\
(\lambda C, e, \sharp a :: p, s) & \rightarrow & (C, \sharp a :: e, p, s) & (4.6)\\
(\lambda C, e, M(a) :: p, s) & \rightarrow & (\lambda C, e, p, s[a \leftarrow <\lambda C, e>]) & (4.7)\\
(\underline{1}, \sharp a :: e_1, p, s) & \rightarrow & (C_2, e_2, M(a) :: p, s) & (4.8)\\
 & & \text{où } <C_2, e_2> = s(a) & \\
(\underline{n+1}, \sharp a :: e, p, s) & \rightarrow & (\underline{n}, e, p, s) & (4.9)
\end{array}
$$

FIG. 4.2 – *La machine KP*

utilisée, et l'opération d'accès aux fermetures, définie par la fonctionnelle suivante :

$$s[a \leftarrow f] = \lambda a'.\ \textbf{si}\ a = a'\ \textbf{alors}\ f\ \textbf{sinon}\ s(a').$$

Syntaxiquement, on distingue les adresses des références. Une adresse est notée a et sa référence est notée $\sharp a$. Cette distinction n'est qu'une nuance entre l'adresse vue comme l'emplacement où on a stocké une fermeture, et l'adresse vue comme un représentant de la fermeture à laquelle elle fait référence. Ainsi, dans les environnements et la pile, on considère les références (les fermetures représentées par ces adresses), alors que dans les marques, on considère l'emplacement physique (l'adresse à laquelle on va devoir faire la mise-à-jour, indépendamment de la fermeture à laquelle elle fait référence).

La nouvelle machine est décrite par le système de réécriture de la figure 4.2.

Notons que dans la règle 4.5, l'opération sur l'état mémoire $s[a \leftarrow <C_2, e>]$ représente une création d'emplacement mémoire (a = nouveau(s)), mais que l'opération $s[a \leftarrow <\lambda C, e>]$ de la règle 4.7 représente une modification de l'état mémoire à l'adresse a.

La règle 4.8 place une marque sur la pile au moment où la fermeture dépilée va être évaluée, de telle sorte que l'évaluation ait lieu en dehors de tout contexte. Lorsque la marque sera dépilée, il n'y aura plus qu'à mettre la mémoire à jour (règle 4.7) et à continuer l'évaluation. Ainsi, lors de la duplication des environnements (règle 4.5), ce ne sont pas des fermetures, mais leurs références, que l'on dupliquera. Par conséquent, une fermeture déjà évaluée n'aura pas à être évaluée une nouvelle fois.

Pierre Crégut propose dans sa thèse des optimisations pour cette machine. Ces optimisations ne sont pas décrites ici, car je ne les ai pas mises en œuvre dans la UP-machine. De plus, il propose d'autres machines nommées KN et KNP qui permettent de calculer la forme normale des termes. Le lecteur intéressé pourra se référer à [Cré91] pour plus de détails.

Chapitre 5

La UP-machine

Ce chapitre décrit l'implantation de la UP-machine, une version de la U-machine avec appel paresseux réalisé grâce au partage des fermetures sur le même principe que la KP-machine proposée par Pierre Crégut [Cré91]. Dans un premier temps, je présente la U-machine, puis j'explique comment j'ai obtenu la nouvelle machine à partir de la précédente et des idées proposées par P. Crégut.

5.1 Évaluation faible et évaluation forte

Dans le chapitre précédent, j'ai présenté le travail de Pierre Crégut sur la machine de Krivine qui fait de l'évaluation faible des termes du λ-calcul, c'est-à-dire que seuls les radicaux de tête du terme donné peuvent être réduits. Cette machine a été utilisée ensuite comme base pour réaliser les machines KN et KNP qui calculent la forme normale (faible) des termes. La U-machine, par contre, est vouée à faire de l'évaluation forte des termes, c'est-à-dire à calculer la forme normale forte des termes du λ-calcul.

L'évaluation faible est le mécanisme utilisé par les langages de programmation fonctionnelle. En effet, lorsqu'on écrit un programme, tout le code n'est pas directement évalué ; si on évalue les appels de fonctions (radicaux), par contre on n'évalue pas les fonctions lorsqu'elles ne sont pas appliquées (abstractions hors radical), ni les appels qu'elles contiennent. Pour plus de détails sur les principes des langages fonctionnels, on pourra se référer à [PJ87].

L'évaluation forte est le mécanisme utilisé pour les systèmes de preuve ou la simplification de programme. En effet, dans ce cadre, on a besoin de parcourir l'ensemble du terme. Les machines U et UP sont donc tournées vers cet horizon.

5.2 La U-machine

La U-machine est décrite dans [LR93]. C'est une machine à environnements qui évalue des termes du λ-calcul grâce au mécanisme de substitutions explicites du $\lambda\upsilon$-calcul.

À chaque règle du $\lambda\upsilon$-calcul correspond une règle de la U-machine, à l'exception des règles (Lambda) et (Beta) qui sont liées à une seule règle de la machine à environnement. D'autre part, la U-machine contient des règles d'inférence qui

état	=	terme $\times$ environnement $\times$ pile
environnement	=	liste de (fermetures $\cup \uparrow$) $\times$ $\mathbb{N}$
pile	=	liste de fermetures
fermeture	=	terme $\times$ environnement

FIG. 5.1 – *Structures de la U-machine*

$$
\begin{array}{rcll}
(a\ b, e, p) & \xrightarrow[U]{} & (a, e, <b, e>:: p) & \text{(APP)} \\
(\lambda a, e, <b, e'>:: p) & \xrightarrow[U]{} & (a, \mathbf{Lift\_env}(e)\ \oplus\ [(<b, e'>, 0)], p) & \text{(LBA-BET)} \\
(\underline{1}, (c, i+1) :: e, p) & \xrightarrow[U]{} & (\underline{1}, e, p) & \text{(FVARLIFT)} \\
(\underline{n+1}), (c, i+1) :: e, p) & \xrightarrow[U]{} & (\underline{n}, (c, i) :: (\uparrow, 0) :: e, p) & \text{(RVARLIFT")} \\
(\underline{1}, (<a, e>, 0) :: e', p) & \xrightarrow[U]{} & (a, e \oplus e', p) & \text{(FVAR)} \\
(\underline{n+1}, (<a, e>, 0) :: e', p) & \xrightarrow[U]{} & (\underline{n}, e', p) & \text{(RVAR)} \\
(\underline{n}, (\uparrow, 0) :: e, p) & \xrightarrow[U]{} & (\underline{n+1}, e, p) & \text{(VARSHIFT)}
\end{array}
$$

FIG. 5.2 – *La U-machine*

permettent d'obtenir la forme normale des termes (pour les appels récursifs de la machine).

Les structures de la machine sont données dans la figure 5.1. Les termes sont des termes purs du λ-calcul (i.e. sans fermetures) en notation de De Bruijn. L'entier qui apparaît dans les environnements traduit le nombre de *lift* ($\Uparrow$) appliqués.

La U-machine est décrite par le système de règles de la figure 5.2, où **Lift_env**, équivalent du *lift* du λv-calcul ($\Uparrow$), est défini par

$$
\begin{array}{rcl}
\mathbf{Lift\_env}([\,]) & \to & [\,] \\
\mathbf{Lift\_env}((c, i) :: e) & \to & (c, i+1) :: \mathbf{Lift\_env}(e)
\end{array}
$$

et l'opérateur $\oplus$ concatène deux environnements.

La U-machine s'arrête dans un état de la forme $(\lambda a, e, [\,])$ ou $(\underline{n}, [\,], p)$. Pour le calcul de la forme normale des termes, il est nécessaire d'appeler récursivement la U-machine. Deux règles d'inférence définissant nf permettent cet appel récursif (voir figure 5.3).

$$\frac{(a,e,[\,]) \xrightarrow[U]{*} (\lambda b, e', [\,]) \qquad (b, \mathbf{Lift\_env}(e')) \xrightarrow[nf]{} c}{(a,e) \xrightarrow[nf]{} \lambda c} \quad \text{(L)}$$

$$\frac{(a,e,[\,]) \xrightarrow[U]{*} (\underline{n}, [\,], [<b_1,e_1>;\ldots;<b_q,e_q>]) \qquad <b_i,e_i> \xrightarrow[nf]{} c_i}{(a,e) \xrightarrow[nf]{} \underline{n}\ c_1 \ldots c_q} \quad \text{(V)}$$

FIG. 5.3 – *Règles de la U-machine pour l'obtention de la forme normale des termes*

La règle (L) force le calcul sous les λ n'appartenant à aucun radical (on fait de l'évaluation forte), et la règle (V) permet de calculer les formes normales des arguments qui sont dans la pile, ces arguments n'appartenant non plus à aucun radical, lorsqu'on a atteint la condition d'arrêt, variable dans un environnement vide.

La preuve de la correction de cette machine peut être trouvée dans [LR93].

5.3 Réalisation de la UP-machine

On souhaite obtenir une machine équivalente à la U-machine, mais dont la stratégie d'évaluation des termes soit paresseuse, grâce à l'implantation du partage des arguments des fonctions (fermetures). Pour cela, on s'inspire du travail de Pierre Crégut (voir chapitre 4) sur la machine de Krivine.

Pour le passage de la machine de Krivine à la machine KP, on avait introduit des marques qui permettaient d'isoler l'évaluation des fermetures lors de leur utilisation en tant qu'arguments, afin de pouvoir réutiliser ultérieurement, dans un autre contexte, le résultat stocké dans un état mémoire. Ici, on met en œuvre le même principe ; mais il est compliqué par le fait qu'on cherche à présent à calculer la forme normale forte des termes.

5.3.1 Définition de UP

Comme pour le passage de la machine de Krivine à la machine KP, on effectue sur la U-machine les modifications suivantes :

- on ajoute un champ état mémoire sur les états, contenant des adresses auxquelles sont associées des fermetures,

- on remplace les fermetures par leur référence dans l'état mémoire,

état	=	terme × environnement × pile × état mémoire
terme	=	terme pur du λ-calcul en notation de De Bruijn
environnement	=	liste de (@fermeture ∪ ↑) × $\mathbb{N}$
pile	=	liste de (@fermeture ∪ M(@fermeture × environnement))
état mémoire	=	liste d'associations [adresse ← fermeture]
fermeture	=	terme × environnement

FIG. 5.4 – *Structures de la UP-machine*

- on introduit des marques pour isoler de leur contexte l'évaluation des arguments, et mémoriser l'adresse à mettre à jour une fois cette évaluation terminée.

Les deux premières modifications ne présentent aucune difficulté. La troisième demande un peu plus de réflexion.

En effet, considérons la règle (FVAR) de la U-machine

$$(\underline{1}, (<a, e>, 0) :: e', p) \xrightarrow[U]{} (a, e \oplus e', p)$$

contrairement à ce qui se passait pour la machine de Krivine, on ne remplace pas ici un environnement par un autre, mais on concatène deux environnements ($e \oplus e'$) car on a besoin de conserver l'ancien environnement[1]. Or, on aimerait dans un premier temps réaliser l'évaluation de la fermeture $<a, e>$ indépendamment de l'environnement e' et de l'état de la pile p. C'est pourquoi les marques que l'on introduit pour la machine UP sont différentes de celles introduites pour la machine KP ; elles stockent à présent non seulement une adresse, mais aussi un environnement, l'environnement e', que l'on devra restaurer au moment où l'on dépilera la marque qui le contient.

On obtient alors un système de réécriture dont les structures sont données dans la figure 5.4 et les règles dans la figure 5.5 où **Lift_env** est défini par

$$\begin{array}{rcl} \mathbf{Lift\_env}([\,]) & \rightarrow & [\,] \\ \mathbf{Lift\_env}((c, i) :: e) & \rightarrow & (c, i+1) :: \mathbf{Lift\_env}(e) \end{array}$$

et l'opérateur $\oplus$ concatène deux environnements..

Les opérations de base sur les états mémoire et les adresses sont définies par :

- Soit s un état mémoire, a une adresse et f une fermeture,

$$s[a \leftarrow f] = \lambda a'.\ \mathbf{si}\ a = a'\ \mathbf{alors}\ f\ \mathbf{sinon}\ s(a')$$

1. On rappelle qu'on calcule la forme normale forte des termes, alors que la machine de Krivine et la machine KP ne calculaient que la forme normale de tête.

$$
\begin{array}{rcll}
(b\ c, e, p, s) & \xrightarrow[UP]{} & (b, e, \sharp a :: p, s[a \leftarrow <c,e>]) & \text{(APP)} \\
 & & \text{où } a = \text{nouveau}(s) & \\
(\lambda c, e, \sharp a :: p, s) & \xrightarrow[UP]{} & (c, \mathbf{Lift\_env}(e) \oplus [(\sharp a, 0)], p, s) & \text{(LBA-BET)} \\
(\underline{1}, (c, i+1) :: e, p, s) & \xrightarrow[UP]{} & (\underline{1}, e, p, s) & \text{(FVARLIFT)} \\
(\underline{n+1}, (c, i+1) :: e, p, s) & \xrightarrow[UP]{} & (\underline{n}, (c, i) :: (\uparrow, 0) :: e, p, s) & \text{(RVARLIFT")} \\
(\underline{1}, (\sharp a, 0) :: e', p, s) & \xrightarrow[UP]{} & (c, e, M(a, e') :: p, s) & \text{(FVAR)} \\
 & & \text{où } <c,e> = s(a) & \\
(\underline{n+1}, (\sharp a, 0) :: e, p, s) & \xrightarrow[UP]{} & (\underline{n}, e, p, s) & \text{(RVAR)} \\
(\underline{n}, (\uparrow, 0) :: e, p, s) & \xrightarrow[UP]{} & (\underline{n+1}, e, p, s) & \text{(VARSHIFT)} \\
(\lambda c, e, M(a, e') :: p, s) & \xrightarrow[UP]{} & (\lambda c, e \oplus e', p, s[a \leftarrow <\lambda c, e>]) & \text{(LMK)}
\end{array}
$$

FIG. 5.5 – *La UP-machine*

– la fonction nouveau associe une adresse à un état mémoire, de telle façon que, soit s un état mémoire, nouveau(s) n'appartient pas au domaine de s.

Comme pour la machine KP, la même notation dans les règles (APP) et (LMK) a deux sens nuancés: dans le premier cas, $s[a \leftarrow < c, e >]$ où $a =$ nouveau(s) signifie que l'on crée cette nouvelle adresse (a), et que l'on y place la fermeture $<c, e>$; dans le deuxième cas, il s'agit d'une modification de la fermeture placée à une adresse existante.

Avec la règle (APP), on introduit les nouvelles adresses et on crée une association [adresse$\leftarrow$fermeture] dans l'état mémoire pour stocker la fermeture correspondant à l'argument rencontré. C'est la référence de cette fermeture qui sera partagée lors de la duplication des environnements. Les autres règles sont très proches des règles de la U-machine et ne présentent aucune difficulté, exception faite des règles (FVAR) et (LMK). Ces deux règles permettent de résoudre les problèmes exposés précédemment. La règle (FVAR) isole l'environnement et la pile de l'argument à évaluer, en utilisant la marque dans laquelle sont stockés l'adresse à mettre à jour et l'environnement à restaurer à la fin de l'évaluation de l'argument. Cette mise-à-jour de l'état mémoire et la restauration de l'environnement sont effectuées par la règle (LMK).

5.3.2 Calcul de la forme normale

On doit à présent redéfinir l'opérateur nf qui permet le calcul de la forme normale forte des termes.

Pour la machine U, on avait les conditions d'arrêt suivantes (conditions dans lesquelles on ne peut plus appliquer aucune des règles de U) :

- on se trouve dans un état de la forme $(\lambda a, e, [\,])$, i.e. le λ rencontré ne fait partie d'aucun radical,
- on se trouve dans un état de la forme $(\underline{n}, [\,], p)$, i.e. la forme normale du terme est une application dont le premier terme est une variable.

Dans le premier cas on devait prolonger l'évaluation du terme sous le lambda (règle (L)), et dans le deuxième cas, on devait calculer la forme normale de tous les arguments empilés pour restituer le résultat de l'évaluation qui était une application avec variable en tête (règle (V)).

Avec la machine UP, les conditions d'arrêt restent les mêmes :

- on se trouve dans un état de la forme $(\lambda a, e, [\,], s)$,
- on se trouve dans un état de la forme $(\underline{n}, [\,], p, s)$.

On doit réaliser des règles qui ont le même effet que les règles de la U-machine. Pour le calcul de la forme normale des termes sous les λ ne faisant partie d'aucun radical, la règle (L) se traduit assez systématiquement pour UP (voir figure 5.6).

Par contre, pour le calcul de la forme normale des termes présents dans la pile quand on est dans un état de la forme $(\underline{n}, [\,], p, s)$, les transformations à réaliser ne sont pas aussi simples. En effet, on a dans la pile des marques en plus des adresses de fermetures. Or une marque signifie qu'il faut mettre à jour l'état mémoire, restaurer un environnement qui avait été stocké pour isoler l'évaluation d'un sous-terme, et continuer l'évaluation dans ce nouvel environnement.

On ne peut donc évidemment pas évaluer indépendamment la forme normale de chacune des fermetures empilées, comme cela était fait avec la règle (V) de la U-machine. Il faut procéder à un dépilement progressif. On introduit pour cela l'opérateur $\xrightarrow[dep]{}$ qui permet de retrouver la forme normale du terme représenté par un état de la forme $(\underline{n}, [\,], p, s')$ (voir figure 5.6).

Le but de dep est de transformer le couple (terme, pile) en un terme correct en forme normale. La pile contenant des adresses et des marques, l'idée est de réaliser deux règles pour la définition de dep : la première pour dépiler une adresse, la deuxième pour dépiler une marque. On obtient alors le système de la figure 5.7.

La règle (DEPADR) signifie que lorsque le sommet de pile est une adresse, b étant la partie du terme déjà calculée, alors il suffit de calculer la forme normale du terme qui est référencé en sommet de pile, pour en déduire le nouveau terme obtenu par application des deux sous-termes.

La règle (DEPMK) signifie que lorsque le sommet de pile est une marque, alors on doit calculer la forme normale du terme obtenu (champ terme des états de dep) dans le contexte restauré et mettre à jour l'état mémoire. dep s'arrête après utilisation de cette règle, puisque la pile devient alors vide.

Il est à noter que le terme b de la règle (DEPADR) est une variable ou une application, mais en aucun cas une abstraction. En effet, b a été obtenu par applications successives de la règle (DEPADR) (car la règle (DEPMK) provoque

$$\frac{(b,e,p,s) \xrightarrow[UP]{*} (\lambda c, e', [\,], s') \qquad (c, \mathbf{Lift\_env}(e'), [\,], s') \xrightarrow[nf]{} d}{(b,e,p,s) \xrightarrow[nf]{} \lambda d} \quad \text{(L)}$$

$$\frac{(b,e,p,s) \xrightarrow[UP]{*} (\underline{n}, [\,], p, s') \qquad (\underline{n}, p, s') \xrightarrow[dep]{*} (d, [\,], s'')}{(b,e,p,s) \xrightarrow[nf]{} d} \quad \text{(V)}$$

FIG. 5.6 – *Règles de la UP-machine pour le calcul de la forme normale des termes*

$$\frac{(c,e,[\,],s) \xrightarrow[nf]{} d \text{ où } <c,e>= s(a)}{(b, \sharp a :: p, s) \xrightarrow[dep]{} (b\ d, p, s[a \leftarrow <d,[\,]>])} \quad \text{(DEPADR)}$$

$$\frac{(b,e,p,s[a \leftarrow <b,[\,]>]) \xrightarrow[nf]{} d}{(b, M(a,e) :: p, s) \xrightarrow[dep]{} (d, [\,], s[a \leftarrow <b,[\,]>])} \quad \text{(DEPMK)}$$

FIG. 5.7 – *définition de l'opérateur dep*

la fin du calcul par *dep* en rendant un environnement vide), qui ne fait qu'appliquer une suite de termes. Le premier terme de l'application étant une variable ($\underline{n}$) et les suivants étant en forme normale, on est donc assuré que le terme $(b\ d)$ est en forme normale.

5.4 Une optimisation de la UP-machine

On peut remarquer que lorsqu'une fermeture a été évaluée une première fois, ses utilisations ultérieures se résumeront à :

1. poser la marque sur la pile à l'appel
2. tenter d'évaluer le terme. On se rend compte qu'on est tout de suite dans l'obligation de prendre une fermeture en sommet de pile
3. donc, retirer la marque en recopiant la fermeture sur elle même à l'adresse qu'elle contient, et continuer l'évaluation

On aura donc utilisé trois pas de réécriture inutilement. L'idée serait donc d'ajouter dans les états mémoire un indicateur à deux états : le premier état indiquant que la fermeture n'a pas encore été évaluée (N), et le second qu'elle l'a été (E).

Le système obtenu est alors le suivant :

$$
\begin{array}{rcll}
(b\ c, e, p, s) & \xrightarrow[UP]{} & (b, e, \sharp a :: p, s[a \leftarrow <c, e, N>]) & \text{(APP)} \\
 & & \text{où } a = \text{nouveau}(s) & \\
(\lambda c, e, \sharp a :: p, s) & \xrightarrow[UP]{} & (c, \mathbf{Lift\_env}(e) \oplus [(\sharp a, 0)], p, s) & \text{(LBA-BET)} \\
(\underline{1}, (c, i+1) :: e, p, s) & \xrightarrow[UP]{} & (\underline{1}, e, p, s) & \text{(FVARLIFT)} \\
(\underline{n+1}, (c, i+1) :: e, p, s) & \xrightarrow[UP]{} & (\underline{n}, (c, i) :: (\uparrow, 0) :: e, p, s) & \text{(RVARLIFT")} \\
(\underline{1}, (\sharp a, 0) :: e', p, s) & \xrightarrow[UP]{} & \text{si } s(a) = <c, e, N> & \\
 & & \text{alors } (c, e, M(a, e') :: p, s) & \\
 & & \text{sinon } (c, e \oplus e', p, s) & \text{(FVAR)} \\
(\underline{n+1}, (\sharp a, 0) :: e, p, s) & \xrightarrow[UP]{} & (\underline{n}, e, p, s) & \text{(RVAR)} \\
(\underline{n}, (\uparrow, 0) :: e, p, s) & \xrightarrow[UP]{} & (\underline{n+1}, e, p, s) & \text{(VARSHIFT)} \\
(\lambda c, e, M(a, e') :: p, s) & \xrightarrow[UP]{} & (\lambda c, e \oplus e', p, s[a \leftarrow <\lambda c, e, E>]) & \text{(LMK)}
\end{array}
$$

$$
\frac{(b, e, p, s) \xrightarrow[UP]{*} (\lambda c, e', [\,], s') \qquad (c, \mathbf{Lift\_env}(e'), [\,], s') \xrightarrow[nf]{} d}{(b, e, p, s) \xrightarrow[nf]{} \lambda d} \text{ (L)}
$$

$$
\frac{(b, e, p, s) \xrightarrow[UP]{*} (\underline{n}, [\,], p, s') \qquad (\underline{n}, p, s') \xrightarrow[dep]{*} (d, [\,], s'')}{(b, e, p, s) \xrightarrow[nf]{} d} \text{ (V)}
$$

$$\frac{(c,e,s) \xrightarrow[nf]{} d \text{ où } <c,e,\_>= s(a)}{(b,\sharp a :: p, s) \xrightarrow[dep]{} (b\ d, p, s[a \leftarrow <d,[\,],E>])} \quad \text{(DEPADR)}$$

$$\frac{(b,e,p,s[a \leftarrow <b,[\,],E>]) \xrightarrow[nf]{} d}{(b, M(a,e) :: p, s) \xrightarrow[dep]{} (d,[\,],s[a \leftarrow <b,[\,],E>])} \quad \text{(DEPMK)}$$

On pourrait se pencher sur d'autres optimisations de la UP-machine pour éliminer de manière plus élégante ces réécritures superflus. Pierre Crégut propose par exemple pour la machine KP une compilation des termes avant l'évaluation afin de marquer les arguments. Une fois un argument évalué, sa marque est otée, de telle manière qu'on ne l'évaluera pas une seconde fois.

5.5 Sûreté de la stratégie de la UP-machine

On doit prouver la sûreté de la stratégie d'évaluation de la machine UP, i.e. montrer que la machine termine pour tout terme normalisable, c'est-à-dire que le nombre de β-réductions appliquées à tout terme normalisable est fini.

On sait que la stratégie de réduction à gauche est sûre ; or la UP-machine fait principalement ses réductions à gauche (on empile tous les arguments et on évalue le sous-terme situé le plus à gauche dans le terme). Cependant, la gestion des marques a une influence sur la stratégie utilisée.

Considérons en effet les règles (V), (DEPADR) et (DEPMK). On peut utiliser un raccourci pour la définition de ces règles, qui donne les nouvelles règles (équivalentes aux trois précédentes) (V') et (V").

$$\frac{\begin{array}{c}(b,e,p,s) \xrightarrow[UP]{*} (\underline{n},[\,],[\sharp a_1;\ldots;\sharp a_q; M(a,e')] \oplus p', s') \\ (c_i,e_i,[\,],s') \xrightarrow[nf]{} d_i \\ (\underline{n}\ d_1 \ldots d_q, e', p', s'[a \leftarrow <\underline{n}\ d_1 \ldots d_q, [\,]>]) \xrightarrow[nf]{} f\end{array}}{(b,e,p,s) \xrightarrow[nf]{} f} \quad (V')$$

$$\frac{(b,e,p,s) \xrightarrow[UP]{*} (\underline{n},[\,],[\sharp a_1;\ldots;\sharp a_q], s') \quad (c_i,e_i,[\,],s') \xrightarrow[nf]{} d_i}{(b,e,p,s) \xrightarrow[nf]{} \underline{n}\ d_1 \ldots d_q} \quad (V'')$$

où $<c_i, e_i>= s(a_i)$ pour $1 \leq i \leq q$.

On peut supposer que, dans le cas de la règle (V'), si $\underline{n}$ se réduit dans l'environnement e' en un terme de la forme λc, et si l'une des fermetures empilées n'est pas normalisable, mais n'est pas appelée non plus dans le terme λc, alors la sûreté de l'évaluation n'est pas assurée. Cependant, on peut montrer qu'il n'est pas possible que la variable de tête du terme se substitue à un autre terme et donc que l'environnement e' a un rôle de renommage uniquement.

En effet, $(\underline{n}, [\,], [\sharp a_1; \ldots; \sharp a_q], s')$ est le résultat de l'évaluation hors contexte de la fermeture placée à l'adresse a par la règle (APP), dont toutes les variables sont déclarées plus haut dans le terme. En particulier, $\underline{n}$ a été déclarée plus haut dans le terme, et au moment de l'application de la règle (APP), le substitut potentiel pour $\underline{n}$ se trouve déjà dans l'environnement e de la fermeture empilée. Par conséquent, $(\underline{n}\ d_1 \ldots d_q, e', p', s')$ se réduira en un terme de la forme $\underline{n'}\ d'_1 \ldots d'q\ d_{q+1} \ldots d_r$ où $\underline{n'}$ est un renommage de la variable $\underline{n}$ et $d'_1 \ldots d'_q$ sont les termes $d_1 \ldots d_q$ dont toutes les variables sont renommées en fonction de leur position dans le terme.

On peut donc en conclure que la stratégie de la UP-machine est sûre.

5.6 Preuve de correction de la UP-machine

Il est important, lorsqu'on propose une telle machine, de prouver sa correction. Prouver la correction de la machine signifie prouver qu'elle est équivalente au système λv, et prouver qu'elle termine, i.e. qu'entre deux β-réductions, elle réalise un nombre fini de réécritures[2].

Dans un premier temps, on montre que les transitions effectuées avec le système UP sont équivalentes à des transitions obtenues par le système λv. Pour cela, on définit une fonction de traduction ξ des états de la machine vers les termes du λv-calcul, comme cela est fait pour la preuve de la U-machine [LR93], et on montre que

$$(a, e, p, s) \xrightarrow[UP]{} (a', e', p', s') \Rightarrow \xi(a, e, p, s) \xrightarrow[\lambda v]{*} \xi(a', e', p', s')$$

Cette partie de la preuve se trouve en annexe D.

On doit ensuite montrer que la fonction nf est équivalente à la réécriture du terme par des règles de λv, i.e.

$$(a, e, s) \xrightarrow[nf]{} b \Rightarrow \xi(a, e, [\,], s) \xrightarrow[\lambda v]{*} b$$

Cette partie de la preuve est compliquée par les définitions croisées de nf et *dep* qui sont un obstacle au raisonnement par induction. Cependant la définition alternative de (V), (DEPMK) et (DEPADR) proposée dans la section précédente sous la forme des deux règles (V) et (V') peut être une piste pour pallier à ce problème. Le problème qui subsiste est de trouver une hypothèse d'induction. Cette partie de la preuve n'a pas pu encore être réalisée.

La dernière partie de la preuve consistera à montrer que le terme rendu par le système après évaluation complète est bien la forme normale du terme de départ, et que l'évaluation termine dans le cas où le terme est normalisable.

2. En effet, on ne peut pas prouver qu'elle termine pour tout terme puisque certains termes ne sont pas normalisables. En d'autres termes on ne peut pas montrer que le nombre de β-réductions est fini.

Chapitre 6

Résultats

Plusieurs tests ont été réalisé pour évaluer les performances de la UP-machine, et les comparer avec celles de la U-machine.

6.1 Tests réalisés

Les mesures qui nous intéressent sont le nombre de réécritures effectuées [1] pour l'évaluation d'un terme et le temps de calcul.

Outre le fait qu'il dépende de la machine sur laquelle on réalise les tests, le temps de calcul est en fait à peu près proportionnel au nombre de réécritures effectuées. On observe que, sur le matériel utilisé [2], la U-machine exécute entre 18 000 et 19 000 réécritures par seconde, alors que la UP-machine exécute de l'ordre de 16 000 à 17 000 réécritures par seconde. La raison de cet écart est le temps utilisé pour les accès aux fermetures en mémoire. On verra plus loin que cet écart de vitesse n'est souvent plus significatif étant donné la grande différence entre le nombre de réécritures réalisées par la machine U et le nombre de réécritures réalisées par la machine UP pour la réduction d'un même terme.

6.1.1 Tests sur les entiers de Church

Les premiers tests réalisés sont ceux proposés par Pierre Crégut dans sa thèse [Cré91]. Il s'agit d'évaluer des termes de complexité croissante afin d'observer le comportement des différentes machines testées. Les entiers de Church sont des termes parfaitement adaptés à ce type de test, puisque pour un entier de Church n, on a n occurrences de la même variable.

Church définit les entiers de la manière suivante :

$$\begin{array}{rcl} \bar{0} & \equiv & \lambda f.\lambda x.x \\ \bar{1} & \equiv & \lambda f.\lambda x.(f\ x) \\ & \vdots & \\ \bar{n} & \equiv & \lambda f.\lambda x.\underbrace{(f\ (f\ (\ldots\ (f}_{n \text{ fois}}\ x)))\ldots) \end{array}$$

1. C'est-à-dire qu'on compte une réécriture pour chaque utilisation d'une des règles présentées

2. Un SUN 4c modèle 4/20

On évaluera les termes $(\bar{2}\ \bar{n}\ I)$, $(\bar{n}\ \bar{2}\ I)$, $(\bar{2}\ \bar{n}\ \bar{2}\ I)$, $(\bar{n}\ \bar{2}\ \bar{2}\ I)$ et $(\bar{2}\ \bar{2}\ \bar{n}\ I)$ où $\bar{n}$ est un entier de Church qui varie, et I l'identité $(I \equiv \lambda x.x)$. Ces termes ont pour particularité de présenter un grand nombre d'occurrences pour chaque variable, ce qui permettra de voir le gain obtenu par le partage dans des cas théoriquement très favorables. On calculera ensuite le n^{ième} successeur du zéro de Church $(church\_successor^n\ \bar{0})$, et enfin on testera la fonction prédécesseur sur les entiers de Church. Ces fonctions sont définies en annexe A.

6.1.2 Tests sur des fonctions classiques

Les tests suivants ont été réalisés sur des fonctions classiques de la programmation fonctionnelle écrites en λ-calcul avec les numéraux de Barendregt. Ces fonctions sont la relation inférieur, la factorielle, l'exposant, et les fonctions de fibonacci et takeushi. Leur définition se trouve en annexe A. L'intérêt de ces tests est d'observer le comportement relatif des deux machines sur des fonctions courantes, mais également d'avoir une idée des performances de la machine par rapport aux temps usuels de calcul de telles fonctions.

6.2 Résultats

Les résultats des différents tests sont présentés sous forme de tableaux, donnant le nombre d'étapes de réécriture nécessaires pour réduire les termes donnés en entrée. Un «-» signifie que le résultat de l'évaluation n'a pas pu être calculé dans un temps raisonnable. La ligne *complexité* affiche, dans les cas où elle a été calculée, l'expression qui donne le nombre de réécritures en fonction de la valeur de n.

6.2.1 Numéraux de Church

Application de numéraux

$\bar{2}\ \bar{n}\ I$

n	U	UP
0	13	14
1	51	57
2	127	126
3	241	229
4	393	366
10	2 103	1 902
complexité	$19n^2 + 19n + 13$	$17n^2 + 18n + 22$

$\bar{n}\ \bar{2}\ I$

n	U	UP
0	8	9
1	41	47
2	127	126
3	363	295
4	963	660
5	2 419	1 449
6	5 843	3 150
9	71 187	31 261

$\bar{2}\ \bar{n}\ \bar{2}\ I$

n	U	UP
1	105	102
2	1 767	947
3	112 627	38 540
4	24 883 963	6 750 443

$\bar{n}\ \bar{2}\ \bar{2}\ I$

n	U	UP
1	165	150
2	1 767	947
3	64 791	21 942
4	35 034 685	8 585 653

$\bar{2}\ \bar{2}\ \bar{n}\ I$

n	U	UP
1	280	222
2	1 767	947
3	6 760	3 438
4	18 757	9 495

Successeur de Church

n	U	UP
1	22	26
2	68	68
3	163	138
4	319	236
5	548	362
6	862	516
7	1 273	698
8	1 793	908
9	2 434	1 146
10	3 208	1 412
20	20 903	5 612
50	280 988	35 012
100	2 124 463	140 012
complexité	$2n^3 + 12{,}5n^2 - 5{,}5n + 13$	$14n^2 + 12$

Prédécesseur de Church

n	U	UP
1	225	229
2	747	577
3	1 656	1 018
4	3 088	1 533
5	5 157	2 122
6	7 977	2 785
7	11 662	3 522
8	16 326	4 333
9	22 083	5 218
10	29 047	6 177
20	190 152	19 837
50	2 606 067	105 217
100	19 914 592	395 517
complexité	$19n^3 + 90{,}5n^2 + 95{,}5n + 42$	$37n^2 + 256n - 83$

6.2.2 Fonctions classiques

Relation inférieur

$(inferieur\ \bar{n}\ \bar{3})$

n	U	UP
0	151	150
1	431	439
2	907	823
3	1 574	1 249
4	1 603	1 307
5	1 603	1 307
...	1 603	1 307

$(inferieur\ \bar{n}\ 50)$

n	U	UP
0	151	150
1	431	439
2	907	823
3	1 625	1 299
4	2 631	1 867
5	3 971	2 527
10	17 291	7 207
20	95 431	23 467
40	618 711	83 587
50	1 155 377	126 974
51	1 155 641	127 549
52	1 155 641	127 549
...	1 155 641	127 549
complexité ($n < 50$)	$\frac{23}{3}n^3 + 75n^2 + \frac{592}{3}n + 151$	$46n^2 + 246n + 147$

Factorielle

n	U	UP
0	106	114
1	1 790	1 363
2	13 134	3 980
3	166 905	10 514
4	6 632 776	36 935
5	-	270 454

Exposant

$\bar{2}^{\bar{n}}$

n	U	UP
0	116	124
1	4 440	2 777
2	38 007	7 454
3	491 857	16 239
4	11 279 265	35 215
5	-	83 316
6	-	225 482

$\bar{n}^{\bar{2}}$

n	U	UP
0	293	283
1	7 084	3 170
2	38 007	7 454
3	122 062	14 134
4	306 045	24 380
5	660 203	38 306
6	1 284 429	56 770
10	9 909 763	190 406

Fonction de Fibonacci

n	U	UP
0	425	400
1	1 029	916
2	7 876	4 347
3	21 614	8 616
4	97 703	17 490
5	336 854	32 423
6	1 669 794	59 733
7	8 160 558	108 235
8	56 526 424	196 311

Fonction de Takeushi

n	m	p	U	UP
2	3	4	1 959	1 525
2	4	3	1 948	1 512
3	2	4	322 711	19 740
3	4	2	3 193	2 193
4	2	3	178 748	18 177
4	3	2	37 059	10 004
6	4	2	-	181 232

6.3 Commentaires

Les nombreux tests réalisés montrent, sans contestation possible, l'intérêt du partage qui bien souvent permet de gagner un, voire plusieurs degrés de complexité (*cf* Fibonacci, exposant) dans l'évaluation des suites de termes. Il est à noter que même dans les cas où le gain ne se compte pas en degrés de complexité, la version avec partage est meilleure que la version sans partage, exception faite, bien sûr, de certains cas très particuliers, notamment où les variables ont très peu d'occurrences. Enfin, il est intéressant de constater que les performances comparées des deux machines sur des termes couramment utilisés dans le cadre de la programmation fonctionnelle montrent un net avantage pour la UP-machine, même s'il est important de rappeler qu'on ne se place pas dans le cadre de la programmation fonctionnelle (puisqu'on fait de l'évaluation forte). En particulier on a pu réduire certains termes dans un temps raisonnable avec la UP-machine quand la U-machine se faisait longuement attendre.

Toutefois, de façon générale, malgré l'amélioration par rapport à la U-machine, on peut dire que les temps de calcul de la UP-machine restent longs, voire prohibitifs. Il est donc important de se tourner soit vers des optimisations de la UP-machine, soit vers d'autres techniques d'évaluation, comme par exemple les machines à réducteurs de graphes.

D'autre part, il est important de souligner que la UP-machine est programmée pour l'instant avec un langage de haut niveau (caml-light, voir annexe C pour les sources), et utilise le mécanisme d'accès aux références de caml-light, ainsi que son «ramasse-miettes». Pour une implantation de bas niveau de la machine, il faudrait donc réaliser des fonctions de création, d'accès, et de ramasse-miettes propres à la machine, ce qui rajoutera des étapes de réécriture supplémentaires. Cependant, au vu des résultats en temps de calcul de la UP-machine, on peut penser conserver un gain important d'une machine à l'autre, même à un bas niveau de programmation.

6.4 Résultats de la version optimisée

Les tests réalisés sur la version optimisée (voir section 5.4) de la UP-machine sont moins nombreux que ceux qui ont été réalisés sur la machine initiale et n'ont pas été édités. Cependant, ils mettent en évidence un léger gain en performances de 3 à 6 % du nombre de réécritures sur les termes testés.

Chapitre 7

Conclusion

J'ai proposé dans ce document une nouvelle machine à environnements, nommée UP, basée sur le système $\lambda\upsilon$, sur les techniques de réduction de la U-machine, et sur le partage des fermetures, auparavant proposé par Pierre Crégut. Cette nouvelle machine montre des résultats bien meilleurs que ceux de la U-machine, ce qui est un point très positif de ce travail. En particulier, on obtient une diminution très conséquente de la complexité de l'évaluation de certaines suites de termes. Cependant, elle repose encore sur des intuitions et, en particulier, les preuves de correction et de terminaison restent à achever. D'autre part, même si je n'ai malheureusement pas pu comparer les performances de la UP-machine avec les performances d'autres machines calculant la forme normale forte des termes (U-machine mise à part) la complexité de la réduction de nombreux termes me semble encore élevée. C'est pourquoi il sera à mon avis nécessaire de poursuivre les recherches vers des machines encore plus performantes, pourquoi pas tournées vers des améliorations de la UP-machine, ou encore vers d'autres techniques d'évaluation, comme la réduction de graphes.

Néanmoins, selon moi, ce travail confirme l'intérêt qu'il faut continuer à porter vers l'introduction des notions de partage dans la réduction des termes, malgré les difficultés supplémentaires que cela engendre au niveau théorique.

Je tiens à remercier Jocelyne Rouyer et Pierre Lescanne
qui m'ont encadré durant cette période de stage,
ainsi que tous les membres de mon jury.
Je remercie enfin tous ceux qui, techniquement ou moralement,
m'ont aidé durant cette période.

Bibliographie

[ACCL90] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. Technical Report 54, Digital Systems Research Center, February 1990.

[ASU91] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs - Principes, techniques et outils*. InterEditions, 1991.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[CHL92] P.-L. Curien, Th. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. RR 1617, INRIA, Rocquencourt, February 1992.

[Chu32a] Alonzo Church. A set of postulates for the foundation of logics. *Annals of Math*, 33:346–366, 1932.

[Chu32b] Alonzo Church. A set of postulates for the foundation of logics. *Annals of Math*, 34:839–864, 1932.

[Cré91] P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université de PARIS 07, 1991.

[DB72] N. G. De Bruijn. λ-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, page 34, 1972.

[HL89] Th. Hardin and J.-J. Lévy. A confluent calculus of substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, Izu, 1989.

[Kri85] J.-L. Krivine. Un interpréteur pour le λ-calcul. Notes de cours de DEA, Paris, 1985.

[Lal90] R. Lalement. *Logique, Réduction, résolution*. Etudes et recherches en informatique. Masson, Paris, 1990.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[LR93] P. Lescanne and J. Rouyer. The calculus of explicit substitutions $\lambda\upsilon$. Internal report, December 1993.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall, Inc., 1987.

[Río93] A. Ríos. *Contributions à l'étude des λ-calculs avec des substitutions explicites*. Thèse de Doctorat d'Université, U. Paris VII, 1993.

Annexe A

Définition des termes testés

A.1 Termes divers

SUCC, **PRED** et **ZERO** définis ici sont le successeur, le prédécesseur et le test à zéro de Barendregt. On utilise également le combinateur de point fixe **Y** pour programmer les appels récursifs. En effet, **Y** vérifie la propriété importante suivante :

$$\forall H,\ \mathbf{Y}\ H \rightarrow H\ (\mathbf{Y}\ H)$$

$$\begin{array}{rcl}
\mathbf{TRUE} & \equiv & \lambda x.\lambda y.x \\
\mathbf{FALSE} & \equiv & \lambda x.\lambda y.y \\
\mathbf{SUCC} & \equiv & \lambda x.\lambda y.(y\ \mathbf{FALSE}\ x) \\
\mathbf{PRED} & \equiv & \lambda x.(x\ \mathbf{FALSE}) \\
\mathbf{ZERO} & \equiv & \lambda x.(x\ \mathbf{TRUE}) \\
\mathbf{IF}\ A\ \mathbf{THEN}\ B\ \mathbf{ELSE}\ C & \equiv & A\ B\ C \\
\mathbf{LETREC}\ v = B\ \mathbf{IN}\ C & \equiv & (\lambda v.C\ (\mathbf{Y}\ (\lambda v.B))) \\
\mathbf{LET}\ v = B\ \mathbf{IN}\ C & \equiv & (\lambda v.C\ B) \\
\mathbf{Y} & \equiv & \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x))) \\
\mathbf{FUN}\ x \rightarrow T & \equiv & \lambda x.T
\end{array}$$

A.2 Numéraux

A.2.1 Numéraux de Barendregt

$$\begin{array}{rcl} \bar{0} & \equiv & \lambda x.x \\ \bar{n} & \equiv & \lambda x.(x\ \mathbf{FALSE}\ (\overline{n-1})) \end{array}$$

A.2.2 Numéraux de Church

Définition

$$\begin{array}{rcl} \bar{0} & \equiv & \lambda f.\lambda x.x \\ \bar{1} & \equiv & \lambda f.\lambda x.(f\ x) \\ & \vdots & \\ \bar{n} & \equiv & \lambda f.\lambda x.\underbrace{(f\ (f\ (\ldots\ (f}_{n \text{ fois}}\ x)))\ldots) \end{array}$$

Successeur et prédecesseur de Church

LET $church\_successor =$ **FUN** $n \rightarrow$ **FUN** $x \rightarrow$ **FUN** $y \rightarrow (x\ (n\ x\ y))$
IN LETREC $H =$ **FUN** $x \rightarrow$ **IF** (**ZERO** x)
THEN $\overline{0_c}$
ELSE ($church\_successor$ (H (**PRED** x)))
IN LET $H\_moins\_un =$ **FUN** $x \rightarrow$ (x **SUCC** $\overline{0_b}$)
IN LET $church\_predecessor =$ **FUN** $x \rightarrow$ (H (**PRED** ($H\_moins\_un\ x$)))
IN ($church\_predecessor\ \overline{n_c}$)

Dans cette définition $\overline{n_c}$ représente le n^{ième} entier de Church et n_b le n^{ième} entier de Barendregt.

A.3 Fonctions diverses

Les fonctions suivantes sont définies sur les entiers de Barendregt.

A.3.1 Relation inférieur

LETREC $inf =$ **FUN** $x \rightarrow$ **FUN** $y \rightarrow$
IF (**ZERO** x)
THEN IF (**ZERO** y)
THEN FALSE
ELSE TRUE
ELSE IF (**ZERO** y)
THEN FALSE
ELSE (inf (**PRED** x) (**PRED** y))
IN ($inf\ \bar{n}\ \bar{m}$)

A.3.2 Factorielle

LETREC *plus* = **FUN** $x \rightarrow$ **FUN** $y \rightarrow$
  IF (**ZERO** x)
  THEN y
  ELSE (*plus* (**PRED** x) (**SUCC** y))
IN LETREC *mult* = **FUN** $x \rightarrow$ **FUN** $y \rightarrow$
  IF (**ZERO** x)
  THEN $\bar{0}$
  ELSE (*plus* y (*mult* (**PRED** x) y))
IN LETREC *fact* = **FUN** $x \rightarrow$
  IF (**ZERO** x)
  THEN $\bar{1}$
  ELSE (*mult* x (*fact* (**PRED** x)))
IN (*fact* $\bar{n}$)

A.3.3 Exposant

LETREC *exp* = **FUN** $x \rightarrow$ **FUN** $y \rightarrow$
  IF (**ZERO** y)
  THEN $\bar{1}$
  ELSE (*mult* x (*exp* x (**PRED** y)))
IN (*exp* $\bar{n}$ $\bar{m}$)

A.3.4 Fonction de Fibonacci

LETREC *moins* = **FUN** $x \rightarrow$ **FUN** $y \rightarrow$
  IF (**ZERO** y)
  THEN x
  ELSE (*moins* (**PRED** x) (**PRED** y))
IN LETREC *fib* = **FUN** $x \rightarrow$
  IF (*inf* x $\bar{2}$)
  THEN $\bar{1}$
  ELSE (*plus* (*fib* (*moins* x $\bar{2}$)) (*fib* (**PRED** x)))
IN (*fib* $\bar{n}$)

A.3.5 Fonction de Takeushi

LETREC *tak* = **FUN** $x \rightarrow$ **FUN** $y \rightarrow$ **FUN** $z \rightarrow$
  IF (*inf* y x)
  THEN (*tak* (*tak* (**PRED** x) y z)(*tak* (**PRED** y) z x)(*tak* (**PRED** z) x y))
  ELSE z
IN (*tak* $\bar{n}$ $\bar{m}$ $\bar{p}$)

Annexe B

U-machine

Le programme suivant est écrit en *CAML light*.

```
(* Type terme : type de representation interne des termes. *)
(* variables en notation de DeBruijn                        *)
type terme = V of int
 | L of terme
 | App of terme * terme ;;


(* Type uenv : environnement de la U-machine *)
type uenv = uenvVide | euCons of (euItem * int) * uenv
and euItem = uShift | f of terme * uenv
and uFerm = ff of terme * uenv ;;

(* Type uPile : pile de la U-machine *)
type uPile == uFerm list ;;


(* Fonction Lift_uenv : uenv -> uenv *)
(* Lift_env de la U-machine          *)
let rec Lift_uenv = function
   uenvVide       -> uenvVide
 | euCons((c,i),e)-> euCons((c,i+1),Lift_uenv(e)) ;;


(* Fonction uenvConcat : (uenv * uenv) -> uenv       *)
(* concatenation des environnements de la U-machine *)
let rec uenvConcat = function
   (uenvVide,e)          -> e
 | (euCons((c,i),e),ee) -> euCons((c,i),uenvConcat(e,ee)) ;;


(* Fonction Umachine :                              *)
(* (terme * uenv * uPile) -> (terme * uenv * uPile) *)
```

```
(* regles de la U-machine.                            *)
let rec Umachine = function
   (App(b,c),e,p)
   -> Umachine(b,e,ff(c,e)::p)
 | (L(c),e,ff(b,ee)::p)
   -> Umachine(c,
            uenvConcat(Lift_uenv(e),euCons((f(b,ee),0),uenvVide)),
            p)
 | (V(1),euCons((f(a,e),0),ee),p)
   -> Umachine(a,uenvConcat(e,ee),p)
 | (V(n),euCons((f(a,e),0),ee),p)
   -> Umachine(V(n-1),ee,p)
 | (V(n),euCons((uShift,0),e),p)
   -> Umachine(V(n+1),e,p)
 | (V(1),euCons((c,i),e),p)
   -> Umachine(V(1),e,p)
 | (V(n),euCons((c,i),e),p)
   -> Umachine(V(n-1),euCons((c,i-1),euCons((uShift,0),e)),p)
 | x -> x ;;


(* Fonction unf : (terme * uenv) -> terme *)
(* regles pour l'appel recursif           *)
let rec unf(a,e) = match Umachine(a,e,[]) with
   (L(c),e,[])       -> L(unf(c,Lift_uenv(e)))
 | (V(n),uenvVide,p) -> let rec map_unf = function
                          [] -> V(n)
                        | ff(b,e)::p -> App(map_unf(p),unf(b,e))
                        in map_unf(rev p)
 | _ -> failwith "impossible" ;;
```

Annexe C

UP-machine

Le programme suivant est écrit en *CAML light*.

```
(* Type terme : type de representation interne des termes. *)
(* variables en notation de DeBruijn                        *)
type terme = V of int
 | L of terme
 | App of terme * terme ;;


(* Type upenv : environnement de la UP-machine *)
type upenv = upenvVide | eupCons of (eupItem * int) * upenv
and eupItem = upShift | A of (upFermeture ref)
and upFermeture = F of (terme * upenv) ;;

(* Type upPile : pile de la UP-machine *)
type upPile = upPileVide | pupCons of pupItem * upPile
and pupItem = AA of (upFermeture ref) | M of ((upFermeture ref) * upenv) ;;


(* Fonction Lift_upenv : upenv -> upenv *)
(* Lift_env de la UP-machine            *)
let rec Lift_upenv = function
   upenvVide       -> upenvVide
 | eupCons((c,i),e)-> eupCons((c,i+1),Lift_upenv(e)) ;;


(* Fonction upenvConcat : (upenv * upenv) -> upenv   *)
(* concatenation des environnements de la UP-machine *)
let rec upenvConcat = function
   (upenvVide,e)        -> e
 | (eupCons((c,i),e),f) -> eupCons((c,i),upenvConcat(e,f)) ;;


(* Fonction trm : upFermeture -> terme      *)
```

```
(* retourne la partie terme d'une fermeture *)
let trm = function F(t,e) -> t ;;

(* Fonction envt : upFermeture -> upenv            *)
(* retourne la partie environnement d'une fermeture *)
let envt = function F(t,e) -> e ;;


(* Fonction UPmachine :                                *)
(* (terme * upenv * upPile) -> (terme * upenv * upPile) *)
(* regles de la UP-machine.                            *)
let rec UPmachine = function
   (App(b,c),e,p)
   -> UPmachine(b,e,pupCons(AA(ref (F(c,e))),p))
 | (L(c),e,pupCons(AA(r),p))
   -> UPmachine(c,upenvConcat(Lift_upenv(e),eupCons((A(r),0),upenvVide)),p)
 | (V(1),eupCons((A(r),0),f),p)
   -> let c = !r
      in
      UPmachine(trm(c),envt(c),pupCons(M(r,f),p))
 | (V(n),eupCons((A(r),0),e),p)
   -> UPmachine(V(n-1),e,p)
 | (V(n),eupCons((upShift,0),e),p)
   -> UPmachine(V(n+1),e,p)
 | (V(1),eupCons((c,i),e),p)
   -> UPmachine(V(1),e,p)
 | (V(n),eupCons((c,i),e),p)
   -> UPmachine(V(n-1),eupCons((c,i-1),eupCons((upShift,0),e)),p)
 | (L(c),e,pupCons(M(r,f),p))
   -> r := F(L(c),e);
      UPmachine(L(c),upenvConcat(e,f),p)
 | x -> x
(* Fonction nf : (terme * upenv) -> terme *)
(* regles pour l'appel recursif           *)
and nf(a,e,p) = match UPmachine(a,e,p) with
   (L(c),f,upPileVide)
   -> L(nf(c,Lift_upenv(f),upPileVide))
 | (V(n),upenvVide,p)
   -> let rec l = function
          (b,pupCons(AA(r),p))
          -> let c = !r
             in begin
                r:= F(nf(trm(c),envt(c),upPileVide),upenvVide);
                l(App(b,trm(!r)),p)
                end
        | (b,pupCons(M(r,e),p))
          -> r := F(b,upenvVide);
```

```
            l(nf(b,e,p),upPileVide)
       | (b,upPileVide) -> b
      in l(V(n),p)
 | _ -> failwith "impossible" ;;
```

Annexe D

Preuve de correction de la UP-machine

Afin de prouver la correction de la UP-machine par rapport au $\lambda\upsilon$-calcul, on définit une fonction de traduction ξ des états vers les termes :

$$\begin{array}{rcl} \xi : \textsf{état} & \rightarrow & \textsf{terme} \\ (b, e, [t_1; \ldots; t_q], s) & \mapsto & b\,[\tau(e,s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \end{array}$$

avec

$$\begin{array}{rcl} \tau : \textsf{env} \times \textsf{état mém} & \rightarrow & \textsf{Liste[Substitutions]} \\ ([\,], s) & \mapsto & [\,] \\ ((c,i) :: e, s) & \mapsto & \rho((c,i),s) :: \tau(e,s) \end{array}$$

$$\begin{array}{rcl} \rho : (\uparrow \cup \textsf{adr ferm}) \times \mathbb{N} \times \textsf{état mém} & \rightarrow & \textsf{Substitutions} \\ ((\sharp a, 0), s = [\ldots; a \leftarrow <b, e>; \ldots]) & \mapsto & b\,[\tau(e,s)]/ \\ ((\uparrow, 0), s) & \mapsto & \uparrow \\ ((c, i+1), s) & \mapsto & \Uparrow(\rho((c,i),s)) \end{array}$$

et

$$\begin{array}{rcl} \eta : (\textsf{adr ferm} \cup \textsf{M}(\textsf{adr ferm} \times \textsf{env})) \times \textsf{état mém} & \rightarrow & \textsf{Substitutions} \cup \textsf{terme} \\ (\sharp a, s = [\ldots; a \leftarrow <b, e>; \ldots]) & \mapsto & b\,[\tau(e,s)] \\ (M(a,e), s) & \mapsto & [\tau(e,s)] \end{array}$$

On étend $\Uparrow$ aux listes de substitutions par $\Uparrow([s_1; \ldots; s_k] = [\Uparrow(s_1); \ldots; \Uparrow(s_k)]$ et on définit @ par $a\,[e@e'] = a\,[e][e']$. Les preuves des lemmes 1 et 2 se font par induction sur e.

Lemme 1 $\tau(e \oplus e', s) = \tau(e,s) @ \tau(e',s)$

Lemme 2 $\tau(\mathbf{Lift\_env}(e), s) = \Uparrow(\tau(e,s))$

Lemme 3 *Toute adresse apparaissant dans la pile ou dans les environnements est une référence à une fermeture.*

> **Preuve :** Seule la règle (APP) introduit de nouvelles adresses ; elle leur associe toujours une fermeture. D'autre part, aucune règle ne détruit des adresses présentes dans l'état mémoire. □

Lemme 4 *Si a est une nouvelle adresse ($a = \mathrm{nouveau}(s)$), alors* $\forall <c', e'>$,

$$\begin{array}{rcl} \tau(e,s) & = & \tau(e, s[a \leftarrow <c', e'>]) \\ \eta(t,s) & = & \eta(t, s[a \leftarrow <c', e'>]) \\ \rho(c,n,s) & = & \rho(c,n, s[a \leftarrow <c', e'>]) \\ \xi(c,e,p,s) & = & \xi(c,e,p, s[a \leftarrow <c', e'>]) \end{array}$$

Preuve : a étant une nouvelle adresse, aucune référence à a n'est présente dans la pile ni dans les environnements. Par conséquent, le résultat rendu par les fonctions de traduction est indépendant de la référence à cette adresse. $\Box$

Proposition 1 *Pour tous les états (a, e, p, s) et (a', e', p', s') de la UP-machine,*

$$(a,e,p,s) \xrightarrow[UP]{} (a',e',p',s') \Rightarrow \xi(a,e,p,s) \xrightarrow[\lambda\upsilon]{*} \xi(a',e',p',s').$$

Preuve : On pose $p = [t_1; \ldots; t_q]$ avec $q \geq 0$. On doit considérer chaque règle de la UP-machine.

- (APP) $(b\ c, e, p, s) \xrightarrow[UP]{} (b, e, \sharp a :: p, s[a \leftarrow < c, e >])$ où $a = nouveau(s)$

$$\begin{array}{rcl} \xi(b\ c, e, p, s) & = & (b\ c)\,[\tau(e,s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ & \xrightarrow[\lambda\upsilon]{} & b\,[\tau(e,s)]\,c\,[\tau(e,s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ & = & b\,[\tau(e,s')]\,c\,[\tau(e,s')]\,\eta(t_1,s')\ldots\eta(t_q,s') \\ & & \text{où } s' = s[a \leftarrow <c,e>] \text{ et } a = \mathrm{nouveau}(s) \text{ (voir lemme 4)} \\ & = & b\,[\tau(e,s')]\,\eta(\sharp a, s')\,\eta(t_1,s')\ldots\eta(t_q,s') \\ & = & \xi(b, e, \sharp a :: p, s[a \leftarrow <c,e>]) \end{array}$$

- (LBA-BET) $(\lambda c, e, \sharp a :: p, s) \xrightarrow[UP]{} (c, \mathbf{Lift\_env}(e) \oplus [(\sharp a, 0)], p, s)$
  - cas où $e = [\,]$:
    $\xi(\lambda c, e, \sharp a :: p, s = [\ldots; a \leftarrow <b, e'>; \ldots])$

$$\begin{array}{rl} = & \lambda c\,[\tau(e,s)]\,\eta(\sharp a, s)\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & \lambda c\,[\tau(e,s)]\,b\,[\tau(e',s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & (\lambda c)\ b\,[\tau(e',s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \text{ car } e = [\,] \\ \xrightarrow[\lambda\upsilon]{} & c\,[\,b\,[\tau(e',s)]/]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & c\,[\rho((\sharp a, 0), s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & c\,[\tau([(\sharp a,0)], s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & \xi(c, \mathbf{Lift\_env}(e) \oplus [(\sharp a, 0)], [t_1; \ldots; t_q], s) \\ & \text{car } e = [\,] \text{ et } \mathbf{Lift\_env}([\,]) = [\,] \end{array}$$

– cas où $e \neq [\,]$
$\xi(\lambda c, e, \sharp a :: p, s = [\ldots; a \leftarrow <b, e'>; \ldots])$

$$\begin{array}{cl} = & \lambda c\,[\tau(e,s)]\,\eta(\sharp a, s)\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & \lambda c\,[\tau(e,s)]\,b\,[\tau(e',s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ \xrightarrow[\lambda v]{} & \lambda(\,c\,[\Uparrow(\tau(e,s))])\,b\,[\tau(e',s)]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ \xrightarrow[\lambda v]{} & c\,[\Uparrow(\tau(e,s))][\,b\,[\tau(e',s)]/]\,\eta(t_1,s)\ldots\eta(t_q,s) \\ = & \xi(c, \mathbf{Lift\_env}(e) \oplus [(\sharp a, 0)], p, s) \\ & \text{(voir lemmes 1 et 2)} \end{array}$$

Pour les règles suivantes, UP ne travaille que sur le sommet de la pile, et la partie inférieure reste inchangée. Il est donc suffisant de prouver

$$(a, e, [\,], s) \xrightarrow[UP]{} (a', e', [\,], s') \Rightarrow \xi(a, e, [\,], s) \xrightarrow[\lambda v]{*} \xi(a', e', [\,], s')$$

pour pouvoir en déduire que la proposition est vraie pour ces règles.

– (FVARLIFT) $(\underline{1}, (c, i+1) :: e, [\,], s) \xrightarrow[UP]{} (\underline{1}, e, p, s)$

$$\begin{array}{rcl} \xi(\underline{1}, (c, i+1) :: e, [\,], s) & = & \underline{1}\,[\tau((c,i+1) :: e, s)] \\ & = & \underline{1}\,[\rho((c,i+1),s) :: \tau(e,s)] \\ & = & \underline{1}\,[\rho((c,i+1),s)][\tau(e,s)] \\ & = & \underline{1}\,[\Uparrow(\rho((c,i),s))][\tau(e,s)] \\ & \xrightarrow[\lambda v]{} & \underline{1}\,[\tau(e,s)] \\ & = & \xi(\underline{1}, e, [\,], s) \end{array}$$

– (RVARLIFT")
$(\underline{n+1}, (c, i+1) :: e, p, s) \xrightarrow[UP]{} (\underline{n}, (c, i) :: (\uparrow, 0) :: e, p, s)$

$$\begin{array}{rcl} \xi(\underline{n+1}, (c, i+1) :: e, [\,], s) & = & \underline{n+1}\,[\tau((c,i+1) :: e, s)] \\ & = & \underline{n+1}\,[\rho((c,i+1),s) :: \tau(e,s)] \\ & = & \underline{n+1}\,[\rho((c,i+1),s)][\tau(e,s)] \\ & = & \underline{n+1}\,[\Uparrow(\rho((c,i),s))][\tau(e,s)] \\ & \xrightarrow[\lambda v]{} & \underline{n}\,[\rho((c,i),s)][\uparrow][\tau(e,s)] \\ & = & \underline{n}\,[\rho((c,i),s)][\rho((\uparrow,0),s)][\tau(e,s)] \\ & = & \underline{n}\,[\rho((c,i),s) :: \rho((\uparrow,0),s) :: \tau(e,s)] \\ & = & \underline{n}\,[\tau((c,i) :: (\uparrow,0) :: e, s)] \\ & = & \xi(\underline{n}, (c,i) :: (\uparrow,0) :: e, [\,], s) \end{array}$$

- (FVAR) $(\underline{1}, (\sharp a, 0) :: e', p, s) \xrightarrow[UP]{} (c, e, M(a, e') :: p, s)$
  où $<c, e> = s(a)$
  $\xi(\underline{1}, (\sharp a, 0) :: e', [\,], s = [\ldots; a \leftarrow <c, e>; \ldots])$
  $$\begin{array}{cl} = & \underline{1}\,[\tau((\sharp a, 0) :: e', s)] \\ = & \underline{1}\,[\rho((\sharp a, 0), s) :: \tau(e', s)] \\ = & \underline{1}\,[\rho((\sharp a, 0), s)][\tau(e', s)] \\ = & \underline{1}\,[\,c\,[\tau(e, s)]/][\tau(e', s)] \\ \xrightarrow[\lambda\upsilon]{} & c\,[\tau(e, s)][\tau(e', s)] \\ = & c\,[\tau(e, s)]\,\eta(M(a, e'), s) \\ = & \xi(c, e, M(a, e'), s) \end{array}$$

- (RVAR) $(\underline{n+1}, (\sharp a, 0) :: e, p, s) \xrightarrow[UP]{} (\underline{n}, e, p, s)$
  $\xi(\underline{n+1}, (\sharp a, 0) :: e, [\,], s = [\ldots; a \leftarrow <b, e>; \ldots])$
  $$\begin{array}{cl} = & \underline{n+1}\,[\tau((\sharp a, 0) :: e, s)] \\ = & \underline{n+1}\,[\rho((\sharp a, 0), s) :: \tau(e, s)] \\ = & \underline{n+1}\,[\rho((\sharp a, 0), s)][\tau(e, s)] \\ = & \underline{n+1}\,[b[\tau(e, s)]/][\tau(e, s)] \\ \xrightarrow[\lambda\upsilon]{} & \underline{n}\,[\tau(e, s)] \\ = & \xi(\underline{n}, e, [\,], s) \end{array}$$

- (VARSHIFT) $(\underline{n}, (\uparrow, 0) :: e, p, s) \xrightarrow[UP]{} (\underline{n+1}, e, p, s)$
  $$\begin{array}{rcl} \xi(\underline{n}, (\uparrow, 0) :: e, [\,], s) & = & \underline{n}\,[\tau((\uparrow, 0) :: e, s)] \\ & = & \underline{n}\,[\rho((\uparrow, 0) :: \tau(e, s)] \\ & = & \underline{n}\,[\rho((\uparrow, 0)][\tau(e, s)] \\ & = & \underline{n}\,[\uparrow][\tau(e, s)] \\ & \xrightarrow[\lambda\upsilon]{} & \underline{n+1}\,[\tau(e, s)] \\ & = & \xi(\underline{n+1}, e, [\,], s) \end{array}$$

- (LMK)
  $(\lambda c, e, M(a, e') :: p, s) \xrightarrow[UP]{} (\lambda c, e \oplus e', p, s[a \leftarrow <\lambda c, e>])$
  Considérons la règle (FVAR) qui a servi à poser la marque qui se trouve en sommet de pile. On raisonne par récurrence sur le nombre de marques empilées entre l'instant où celle ci a été empilée et l'instant où elle est dépilée.
  On suppose dans un premier temps que la marque $M(a, e')$ est la dernière empilée, c'est à dire qu'entre l'utilisation de la règle (FVAR) et l'état courant, on n'a pas utilisé la règle (LMK). Or on a déja montré pour toutes les autres règles que
  $$(a, e, p, s) \xrightarrow[UP]{} (a', e', p', s') \Rightarrow \xi(a, e, p, s) \xrightarrow[\lambda\upsilon]{*} \xi(a', e', p', s').$$

On a suivi le chemin de réduction suivant :

$$\begin{array}{lll} (\underline{1}, (\sharp a, 0) :: e', p, s) & \xrightarrow[UP]{} & (c, e, M(a, e') :: p, s) \\ & \xrightarrow[UP]{*} & (\lambda c', e'', M(a, e') :: p, s') \quad \text{où } <c, e>= s(a) \end{array}$$

donc

$$\xi(c, e, M(a, e') :: p, s) \xrightarrow[\lambda\upsilon]{*} \xi(\lambda c', e'', M(a, e') :: p, s').$$

c'est-à-dire

$$c[\tau(e, s)][\tau(e', s)]\, \eta(t_1, s) \ldots \eta(t_q, s) \xrightarrow[\lambda\upsilon]{*} \lambda c'[\tau(e'', s)][\tau(e', s)]\, \eta(t_1, s) \ldots \eta(t_q, s)$$

donc $c[\tau(e, s)] \xrightarrow[\lambda\upsilon]{*} \lambda c'[\tau(e'', s)]$.

$$\begin{array}{lll} \xi(\lambda c', e'', M(a, e') :: p, s') & = & \lambda c'[\tau(e'', s)]\, \eta(M(a, e'), s)\, \eta(t_1, s) \ldots \eta(t_q, s) \\ & = & \lambda c'[\tau(e'', s)][\tau(e', s)]\, \eta(t_1, s) \ldots \eta(t_q, s) \\ & = & \lambda c'[\tau(e'' \oplus e', s)]\, \eta(t_1, s) \ldots \eta(t_q, s) \\ & = & \xi(\lambda c', e'' \oplus e', p, s) \\ & \xrightarrow[\lambda\upsilon]{*} & \xi(\lambda c', e'' \oplus e', p, s[a \leftarrow <\lambda c', e''>]) \end{array}$$

car on remplace une fermeture de l'état mémoire par une fermeture plus réduite.

On montre ensuite par récurrence sur le nombre de marques empilées que la propriété est vraie pour cette règle.

□