Programmation fonctionnelle et substitutions explicites

Pierre LESCANNE
Centre de Recherche en Informatique de Nancy (CNRS)
INRIA-Lorraine
et GDR Programmation
Campus Scientifique, BP 239,
F54506 Vandœuvre-lès-Nancy, France

email: Pierre.Lescanne@loria.fr

Résumé

Le lambda calcul est l'outil théorique de base pour l'étude des langages de programmation et plus précisément des langages de programmation fonctionnels. La substitution y joue un rôle essentiel, c'est pourquoi nous l'avons étudiée et nous présentons un calcul dit de substitutions explicites qui permet de l'intégrer à l'intérieur du lambda calcul. De la, nous introduisons une machine abstraite, la machine à triade qui sert de substrat aux implantations du lambda calcul et donc aux implantation des langages fonctionnels.
Mots-clés : programmation fonctionnelle, lambda-calcul, types abstraits de données, machines abstraits.

Abstract

Lambda calculus is a theoretical tool for the study of programming languages, more specifically of functional programming languages. Substitution plays a main role there. Here we present a calculus called explicit substitutions which allows us to include substitution into the lambda calculus. Therefore, we introduce an abstract machine, the triad machine which is used to describe implementations of lambda calculus and implementation of functional languages.
Key-words: functional programming, lambda-calculus, abstract data types, abstract machines.

1 La programmation fonctionnelle

Un langage fonctionnel est un langage de programmation où, au lieu de décrire des calculs, on décrit directement les fonctions à calculer. Plus précisément, l'objet de base de ces langages est la fonction : *un programme est une fonction* et une telle fonction (on parle dans ce cas de *fonctionnelle*) peut prendre pour paramètre une fonction et rendre une fonction ; on peut aussi considérer des fonctionnelles plus élaborées qui prennent pour paramètres des fonctionnelles et rendent des fonctions ou des fonctionnelles, et ainsi de suite. La prise en compte de fonctions d'ordre quelconque est un concept important de la programmation fonctionnelle.

Le premier et longtemps le plus populaire des langages fonctionnels est LISP. Il tend à être supplanté par ML[1] qui a sur lui l'avantage d'être *fortement typé*, autrement dit de distinguer les fonctions et les objets selon leur type et d'offrir au programmeur la facilité de construire des types de son choix à bases de records (types sommes) et de produits cartésiens (types produits). C'est une caractéristique fondamentale, car elle permet d'éviter de très nombreuses erreurs de programmation parmi les plus graves et les plus difficiles à mettre en évidence, les fameuses erreurs de type. Ces propriétés font de ML un fantastique langage de prototypage très utilisé dans le milieu de la recherche et inaugure un style de programmation tout-à-fait intéressant. ML est aussi un langage extrêmement commode pour la déduction automatique et c'est cet aspect qui nous intéresse le plus.

1. Le nom ML provient de méta-langage, car à l'origine c'était le méta-langage d'un système de démonstration automatique.

La programmation fonctionnelle a encore un autre avantage, c'est *son aptitude à la parallélisation.* En effet, puisqu'on décrit les fonctions et non les calculs, l'ordinateur et son système ont une grande liberté pour s'organiser et en particulier pour effectuer les calculs en parallèle. Pour cela, le chercheur doit proposer des modèles de calculs qui permettent de comprendre cette parallélisation. L'un de ces modèles est le *λ-calcul.* Le λ-calcul abstrait les principes de la programmation fonctionnelle et constitue son modèle théorique. Il offre un cadre mathématique pour parler de l'implantation des langages fonctionnels et des langages de programmation en général. La substitution y joue un rôle fondamental et dans les modèles classiques elle est décrite à l'extérieur du calcul. Le *calcul de substitutions explicites* que nous allons décrire internalise la manipulations des substitutions.

Dans cet article, nous voudrions présenter le λ-calcul en général, puis le calcul des substitutions explicites, puis les machines abstraites pour implanter les langages fonctionnels.

2 Le lambda-calcul

Considérons la fonction trois qui prend en paramètre une fonction et rend la même fonction appliquée à elle-même trois fois. Par exemple, si on l'applique à la fonction f on obtient la fonction qui à x associe la fonction $f(f(f(x)))$. Cette fonction (cette fonctionnelle) pourrait s'écrire en notation mathématique habituelle :

$$f \mapsto (x \mapsto f(f(f(x))),$$

et en CAML[2] qui est un dialecte de ML, nous l'écririons:

let trois = **function** f ->(**function** x ->f(f(f(x))))

que nous pouvons aussi écrire pour simplifier

let trois f = **function** x ->f(f(f(x)))

ou

let trois f x = f(f(f(x)))

ML ne fait aucune différence entre ces écritures et nous répond dans tous les cas

trois : ('a ->'a) ->'a ->'a = <fun>

ce qui veut dire qu'il accepte *trois* comme correcte et précise que *trois* est une fonction qui prend une fonction de 'a vers 'a et rend une fonction de 'a vers 'a. L'expression ('a ->'a) ->'a ->'a est le *type* de *trois*. L'expression <fun> est la «valeur» de l'objet *trois*, mais comme ML ne sait pas imprimer la valeur d'une fonction, il se contente d'écrire <fun>, en fait il ne sait imprimer que les valeurs des types de données construits à partir des valeurs scalaires comme les réels, les entiers ou les chaînes. CAML connaît aussi la fonction successeur sur les entiers succ : int ->int. On peut à l'aide de *trois* et de *succ* définir une fonction

let plus_trois = trois succ

de type int ->int et très naturellement *plus_trois*, qui applique *succ* trois fois, ajoute trois à un entier et ainsi plus_trois 5 vaut 8.

Le lambda-calcul introduit par Alonzo Church [5, 1] en 1930 formalise le concept de fonction et propose une abstraction de ce que nous venons de présenter. Son but est de fonder la logique sur le concept premier de fonction. Dans le lambda-calcul, il n'y a plus que des fonctions. Il s'est révélé évidemment très utile, par la suite, pour formaliser les fondements de la programmation [10] et concevoir de nouveaux langages de programmation [11]. Le lambda-calcul utilise une notation plus compacte pour noter les fonctions ; ainsi il écrit la fonction *trois* sous la forme $\lambda f \cdot \lambda x \cdot f(f(f(x)))$. De même, la fonction *identité* notée I s'y écrit $\lambda x \cdot x$. La fonction (on dit aussi souvent le *combinateur* dans ce cadre) K qui crée des fonctions constantes est le lambda-terme $\lambda c \cdot (\lambda x \cdot c)$; si on l'applique à la constante k on obtient la fonction $\lambda x \cdot$ k, c'est-à-dire la fonction constante de valeur k. C'est sur le lambda calcul que les chercheurs font leurs premières maquettes de traducteurs de langages fonctionnels, car celui-ci rassemble toutes les caractéristiques de ces langages sans la nécessité de prendre en compte les aspects annexes (fort utiles au demeurant) comme les structures de données ou les entrées-sorties.

Le lambda-calcul manipule des fonctions et ainsi fournit deux constructions de base : l'*application* et l'*abstraction*. L'*application* combine un paramètre et une fonction pour définir un résultat retourné ; elle se note en juxtaposant la fonction, disons F au paramètre, disons A, pour obtenir le terme F A. Ainsi on note

2. Pour une introduction didactique à CAML, nous renvoyons à l'excellent livre [16] et à son compagnon [17].

trois I l'application de la fonction trois au paramètre I. Dans ce cas, I est vue comme une valeur fournie à une fonction, mais puisque dans le lambda-calcul tout est fonction, il n'y a pas de différence entre les fonctions et les valeurs, nous parlerons donc de *lambda-termes* ou plus simplement de *termes*. L'*abstraction* crée un terme qui représente une fonction et se note en préfixant le terme qui représente le corps de la fonction par λ suivi d'un nom de variable. Ainsi si M est un terme $\lambda x \cdot M$ représente la fonction qui à x fait correspondre le terme M. On abstrait le terme $f(f(f(x)))$ en le terme x dans $\lambda x \cdot f(f(f(x)))$. La variable x dans $\lambda x \cdot M$ est dite *liée* et le lambda qui la précède est le lieur. Son nom n'a pas d'importance et elle peut-être renommée (α-conversion). Si la variable x apparaît dans un terme sans y être liée, elle est dite *libre*. Abstraire un terme, puis l'appliquer à un autre terme crée une expression qui est une forme complexe du résultat, pour pouvoir utiliser la puissance du lambda-calcul il faut donc simplifier ce type d'expression ; ainsi, (trois I) ne dit pas ce que «vaut» cette expression, or nous savons qu'itérer trois fois l'identité donne l'identité. Plus précisément on veut qu'une expression $(\lambda x \cdot M)\ N$ se simplifie en l'expression $M[N/x]$ où $[N/x]$ est le terme obtenu en substituant toutes les occurrences libres de x dans M par N. La substitution pose cependant un problème car il faut veiller à ne pas lier des variables libres de N, il faut donc éviter ce que l'on appelle la *capture de variables*. Dans la suite, nous allons proposer des mécanismes de substitutions qui évitent la capture et restent néanmoins simples et efficaces.. Quoi qu'il en soit, on peut au niveau théorique éviter facilement les captures dans la définition de la substitution $[N/x]$ en renommant toutes les variables liées du terme M par des variables nouvelles, à condition de supposer au départ que toutes le variables liées sont différentes. C'est coûteux, mais ça marche ! La règle

$$(\lambda x \cdot M)\ N \rightarrow M[N/x]$$

s'appelle la règle β, c'est le mécanisme de base du lambda-calcul. Un terme de la forme $(\lambda x \cdot M)\ N$ s'appelle un *radical* ou *redex* et un terme qui ne contient aucun radical est appelé une *forme normale*.

Avant de traiter quelques exemples, notons d'abord que toutes les fonctions du lambda-calcul sont monadiques c'est-à-dire qu'elles n'ont qu'un seul paramètre, ça n'est pas gênant car le lambda-calcul manipule des fonctions de fonctions et une fonction de type

$$a \times b - - > c$$

peut-être remplacée par une fonction de type

$$a - - > b - - > c,$$

on appelle cela la *curryfication*. D'autre part, le lambda-calcul utilise des conventions, ainsi nous ne noterons plus le point «.» après la variable qui suit un λ. De plus, pour éviter une avalanche de parenthèses, on suppose que les applications s'associent de la gauche vers la droite, ainsi M N P doit se voir comme M appliqué à N et le résultat appliqué à P, autrement dit c'est une abréviation pour (M N) P. Si l'on veut appliquer plutôt N à P on écrira explicitement les parenthèses, c'est-à-dire M (N P).

Exemple 1 *Cet exemple détaille la mise en forme normale du terme* trois I.

$$\begin{aligned}
\text{trois I} &= (\lambda f\ \lambda x\ f(f(f(x))))\ (\lambda x\ x) \\
&\xrightarrow[\beta]{} \lambda x\ f(f(f(x)))\ [\lambda x\ x/f] \\
&= \lambda x\ (\lambda y\ y)\ ((\lambda z\ z)((\lambda w\ w)\ x)) \\
&\xrightarrow[\beta]{} \lambda x\ y[(\lambda z\ z)\ ((\lambda w\ w)\ x)/y] \\
&= \lambda x\ (\lambda z\ z)\ ((\lambda w\ w)\ x) \\
&\xrightarrow[\beta]{} \lambda x\ z[((\lambda w\ w)\ x)/z] \\
&= \lambda x\ (\lambda w\ w)\ x \\
&\xrightarrow[\beta]{} \lambda x\ w[x/w] \\
&= \lambda x\ x\ =\ \text{I}
\end{aligned}$$

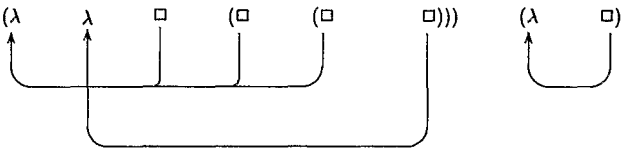


FIG. 1 *Les notations de Bourbaki pour* $(\lambda f\ \lambda x\ f(f(f(x))))\ (\lambda x\ x)$

3 Les indices de de Bruijn

On peut, pour différentes raisons, vouloir éliminer les variables liées. Bourbaki [4] cherchait, dans un souci de fondation des mathématiques, à éliminer tout ce qui de près ou de loin se rapportait à un nombre entier naturel pour éviter d'utiliser des nombres avant de les définir et ainsi ne pas tomber dans un cercle vicieux. Pour cela, il utilise des pointeurs comme illustrés dans la figure 1. De Bruijn [7] propose à la place d'utiliser des indices qui associent à chaque variable le nombre de λ que l'on traverse pour trouver son lieur. Ainsi en utilisant des *indices de de Bruijn* le terme $(\lambda f\ \lambda x\ f(f(f\ x)))\ (\lambda y\ y)$ s'écrit $\lambda(\lambda(\underline{1}(\underline{1}(\underline{1}\ \underline{0}))))\ (\lambda\underline{0})$. En effet, les variables f doivent croiser le λ qui lie x pour trouver leur lieur (d'où la notation $\underline{1}$) alors que pour x et pour y il n'y a pas de lieur à traverser (d'où la notation $\underline{0}$).

Le terme $\lambda k\ \mathsf{K}\ k\ k = \lambda k((\lambda c\lambda x\ c)\ k\ k)$ qui se note $\lambda(\lambda(\lambda\underline{1})\underline{0}\underline{0})$ en indices de de Bruijn se β-réduit en le terme $\lambda k(\lambda x\ k)\ k$ qui se note $\lambda(\lambda(\underline{1})\ \underline{0})$ en indices de de Bruijn. On remarque deux choses. Primo que la même variable (ici k) peut être notée par des indices différents dans le même terme suivant le contexte dans lequel elle se trouve. Secundo, que les indices doivent être décalés (on dit aussi renommés par analogie avec les notations classiques du lambda-calcul) quand on fait une β-réduction.

L'introduction des indices de de Bruijn fait probablement perdre en lisibilité. En revanche, elle permet une notation intrinsèque et canonique des termes. Dans les implantations, elle ne nécessite pas de tables de symbole pour retrouver le lieur d'une variable. Notons que dans [13], nous présentons une autre notation que nous ne reprendrons pas ici ; elle a été introduite par de Bruijn sous le nom de *niveaux* et rend les termes canoniques aussi, tout en étant plus lisibles, car la même variable est associée au même entier dans tout le sous-terme où elle apparaît. Dans ce cas, l'entier qui est associée à une variable est simplement le *niveau* ou la *profondeur* de son lieur. Les indices semblent plus populaires que les niveaux aussi nous les adopterons dans ce qui suit.

4 Le calcul de substitutions explicites $\lambda\upsilon$

La description de la β-réduction par la règle

$$(\lambda x \cdot \mathsf{M})\ \mathsf{N} \rightarrow \mathsf{M}[\mathsf{N}/x]$$

présente l'inconvénient déjà remarqué dès les années 1950 par Haskell Curry [6] de faire appel à une théorie de la substitution qu'il appelle *épithéorie* et qui se situe «autour» de la théorie du lambda-calcul. Pour éviter cet inconvénient, il avait introduit la *logique combinatoire* dont il reconnaissait le caractère peu intuitif si on la compare au lambda-calcul. Les calculs de substitutions explicites répondent au souhait de Curry, car ils visent à réintroduire dans la théorie du lambda-calcul la manipulation des substitutions et proposent ainsi une théorie unique du lambda-calcul sans épithéorie, tout en gardant le côté intuitif du lambda-calcul que n'a pas la logique combinatoire. Dans ce paragraphe, nous introduisons un calcul de substitutions explicites très simple : le calcul $\lambda\upsilon$ (prononcer *lambda-upsilon*) fondé sur les indices de de Bruijn [12]. Il est cependant possible de fonder un calcul de substitutions explicites sur le lambda-calcul classique à variables nommées, comme l'ont montré Rose et Bloo [15].

La première chose à faire dans un calcul de substitutions explicites est d'avoir à sa disposition une notation pour celles-ci. Traditionnellement, on utilise des crochets ; ainsi, si l'on veut affecter la substitution

(B)	$(\lambda M)N$	$\rightarrow$	$M[N/]$
(App)	$(MN)[s]$	$\rightarrow$	$M[s]N[s]$
(Lambda)	$(\lambda M)[s]$	$\rightarrow$	$\lambda(M[\Uparrow(s)])$
(FVar)	$\underline{0}[M/]$	$\rightarrow$	M
(RVar)	$\underline{n+1}[M/]$	$\rightarrow$	$\underline{n}$
(FVarLift)	$\underline{0}[\Uparrow(s)]$	$\rightarrow$	$\underline{0}$
(RVarLift)	$\underline{n+1}[\Uparrow(s)]$	$\rightarrow$	$\underline{n}[s][\uparrow]$
(VarShift)	$\underline{n}[\uparrow]$	$\rightarrow$	$\underline{n+1}$

FIG. 2 *Le système de réécriture $\lambda\upsilon$*

s au terme M, on crée le terme $M[s]$ et l'on dit que $M[s]$ est une *clôture*. Dans ce contexte, la règle β s'écrit :

$$(\lambda M)\ N \rightarrow M[N/]$$

où $N/$ est une substitution particulière qui remplace (ici dans M) l'indice $\underline{0}$ par N et qui décrémente les autres indices de M. Le comportement des substitutions de type / est décrit formellement par les règles de réécriture (LVar) et (RVar) de la figure 2. Pour décrire complètement le comportement des substitutions, il faut savoir les distribuer dans les applications et dans les abstractions. Pour distribuer une substitution s dans une application, il suffit de l'appliquer à chacun de ses membres (règle (App)). Pour faire entrer une substitution dans une abstraction, il faut la modifier (règle (Lambda)) ; pour cela, on lui applique un opérateur $\Uparrow$ appelé *Lift*[3]. $\Uparrow(s)$ est une substitution qui ne modifie pas l'indice $\underline{0}$, car quand s était à l'extérieur de l'abstraction s n'y avait pas accès (règle FVarLift). En revanche, $\Uparrow(s)$ remplace l'indice $\underline{n+1}$ par le terme que s aurait assigné à $\underline{n}$, sachant que comme ce terme est dans un nouveau contexte, il faut lui incrémenter tous ses indices (règle (RVarLift)). Cette incrémentation se fait par l'application d'une substitution appelée *Shift* et notée $\uparrow$ dont le comportement, c'est-à-dire l'incrémentation des indices, est régi par la règle (VarShift).

Les règles de $\lambda\upsilon$ se partagent en deux sous-ensembles, la règle (B) qui élimine un radical en introduisant une substitution et l'ensemble

$$\upsilon = \{(\text{App}), (\text{Lambda}), (\text{FVar}), (\text{RVar}), (\text{FVarLifT}), (\text{RvarLift}), (\text{VarShift})\}$$

composé des règles qui éliminent les substitutions.

Exemple 2 *La suite de réécriture qui suit illustre les indices de de Bruijn et le système $\lambda\upsilon$. Il s'agit de la réduction du terme* trois I. *Dans chaque étape de réécriture, le motif de la règle utilisée (ou de la première règle quand il s'agit d'une suite de réécritures*[4]*) est entouré. La suite de calculs comporte une seule application de la règle (B) suivie d'une élimination des substitutions. Cette suite de règles simule donc une étape de β-réduction. Ce calcul aurait pu être continué jusqu'à la forme normale $\lambda\underline{0}$.*

$$\begin{aligned}
&\boxed{\lambda(\lambda(\underline{1}(\underline{1}\,\underline{0})))\ (\lambda\underline{0})} \xrightarrow[B]{} \boxed{\lambda(\underline{1}(\underline{1}(\underline{1}\,\underline{0})))[(\lambda\underline{0})/]} \\
&\xrightarrow[\upsilon]{} \lambda(\boxed{(\underline{1}(\underline{1}(\underline{1}\,\underline{0})))[\Uparrow(\lambda\underline{0})/]}) \\
&\xrightarrow[\upsilon]{+} \lambda(\boxed{\underline{1}[\Uparrow(\lambda\underline{0})/]}(\underline{1}[\Uparrow(\lambda\underline{0})/)](\underline{1}[\Uparrow(\lambda\underline{0})/)](\underline{0}[\Uparrow(\lambda\underline{0})/)])))) \\
&\xrightarrow[\upsilon]{} \lambda(\boxed{\underline{0}[(\lambda\underline{0})/]}[\uparrow](\underline{1}[\Uparrow(\lambda\underline{0})/)](\underline{1}[\Uparrow(\lambda\underline{0})/)](\underline{0}[\Uparrow(\lambda\underline{0})/)])))) \\
&\xrightarrow[\upsilon]{} \lambda(\boxed{(\lambda\underline{0})[\uparrow]}(\underline{1}[\Uparrow(\lambda\underline{0})/)](\underline{1}[\Uparrow(\lambda\underline{0})/)](\underline{0}[\Uparrow(\lambda\underline{0})/)]))))
\end{aligned}$$

3. La traduction française pourrait en être «*ascenseur*», mais il est douteux qu'elle fasse école.

4. Une suite non vide de réécritures est notée $\xrightarrow{+}$ tandis qu'une suite éventuellement vide de réécritures est notée $\xrightarrow{*}$ ou $\twoheadrightarrow$. La première notation est dans la tradition de la réécriture ou du calcul de relations tandis que la seconde est dans la tradition du lambda-calcul. Le passage à la forme normale se note $\twoheadrightarrow|$.

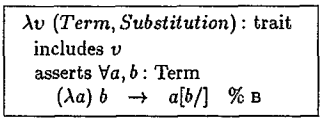


FIG. 3 - *Le trait λv*

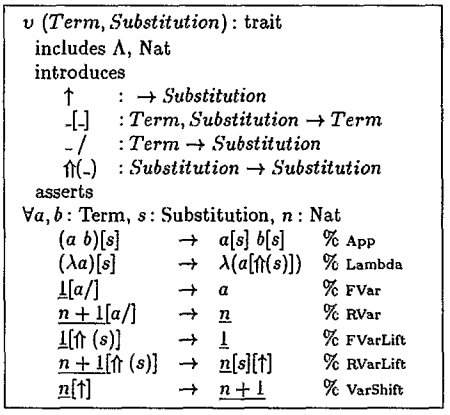


FIG. 4 *Le trait v*

$$\xrightarrow[v]{} \lambda(\lambda(\boxed{\underline{0}[\Uparrow(\uparrow)]})(\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$

$$\xrightarrow[v]{} \lambda(\lambda(\underline{0})(\boxed{\underline{1}[\Uparrow((\lambda\underline{0})/)]}(\boxed{\underline{1}[\Uparrow((\lambda\underline{0})/)]}(\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$

$$\xrightarrow[v]{+} \lambda((\lambda\underline{0})((\lambda\underline{0})((\lambda\underline{0})(\boxed{\underline{0}[\Uparrow((\lambda\underline{0})/)]}))))$$

$$\xrightarrow[v]{} \lambda((\lambda\underline{0})((\lambda\underline{0})((\lambda\underline{0})(\underline{0}))))$$

5 Le lambda calcul comme un type abstrait de données

L'un des aspects qui rend l'approche du lambda-calcul par les substitutions explicites attrayant est le fait que c'est un calcul du premier ordre, c'est-à-dire qu'il n'y a pas de variables de type fonction, et que par

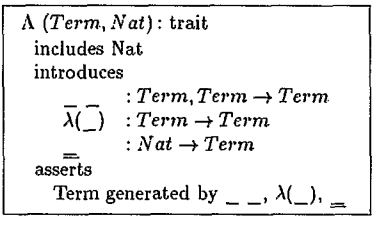


FIG. 5 *Le trait Λ*

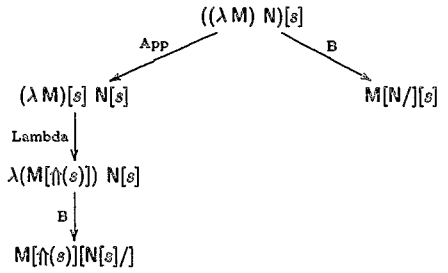


FIG. 6 – *Le lemme de substitution comme paire critique*

conséquent on peut facilement le décrire comme un type abstrait de données. Nous avons choisi le langage partagé de LARCH (*Shared Language* ou LSL) [8]. Son *trait*[5] principal $\lambda\upsilon$ (figure 3) définit la règle (B) tandis que le trait υ (figure 4) décrit le calcul de substitutions proprement dit et le trait Λ (figure 5) auquel il fait appel décrit les termes du lambda-calcul.

6 Quelques propriétés du calcul $\lambda\upsilon$

Ce paragraphe décrit quelques propriétés intéressantes de $\lambda\upsilon$ et d'υ, dont on pourra trouver les démonstrations dans [2]. Le système de réécriture υ termine, c'est-à-dire qu'il n'admet pas de réécriture infinie, dans la terminologie du lambda-calcul on dit aussi que le système est *fortement normalisable*. D'autre part, le système υ *n'admet pas de superposition*, c'est-à-dire qu'il n'y a pas de terme que l'on puisse réécrire à la même position de deux façons différentes. Il est aussi *linéaire à gauche* c'est-a-dire que ses membres gauches ne contiennent que des termes linéaires autrement dit que des termes qui ne possèdent qu'une occurrence de chaque variables. L'absence de superposition et la linéarité à gauche s'appelle l'*orthogonalité*. Enfin tous les motifs sont *couverts*, en effet, si l'on regarde bien toutes les configurations de clôture peuvent-être réécrites ; par exemple la configuration $M[s][t]$ peut-être réécrite, car on peut admettre que le terme $M[s]$ qui est plus simple peut-être réécrit. La terminaison, l'orthogonalité et la couverture sont trois propriétés intéressantes, car elles assurent que l'on peut toujours éliminer complètement les substitutions sans avoir à utiliser une stratégie spécifique. Aucune stratégie ne bouclera, ni ne bloquera sur un terme irréductible. L'orthogonalité [9] nous permet même d'envisager des techniques relativement efficaces.

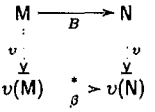
Le système $\lambda\upsilon$ ne termine pas, car il contient le lambda-calcul qui ne termine pas. En effet, on sait que le terme $\Omega = (\lambda x\ xx)\ (\lambda x\ xx)$, en indices de de Bruijn $\lambda(\underline{0}\ \underline{0})\ \lambda(\underline{0}\ \underline{0})$, se β réduit en lui-même[6]. De plus (B) et (App) se superposent et admettent ce que l'on appelle une *paire critique* c'est-à-dire une paire de termes $\langle\ M[N/][s],\ M[\Uparrow(s)][N[s]/]\ \rangle$ issus par réduction du même terme, ici $((\lambda M)\ N)[s]$ (voir figure 6). Les termes $M[N/][s]$ et $M[\Uparrow(s)][N[s]/]$ ne peuvent pas être réduits à la même forme normale et l'égalité $M[N/][s] = M[\Uparrow(s)][N[s]/]$ ne peut pas être prouvée comme une égalité de la théorie $\lambda\upsilon$. Cependant cette égalité peut-être prouvée par récurrence pour tous les termes de Λ et même tous les termes sans variables (termes clos) contenant des clôtures. On l'appelle souvent le *lemme de substitutions*, car elle correspond à un lemme fondamental du lambda-calcul classique qui indique comment on intervertit des substitutions. Il est intéressant de noter que ce lemme a une origine tout-à-faire naturelle du point de vue de la réécriture.

Comme conséquence du lemme de substitution on prouve un *lemme de projection* qui affirme que si

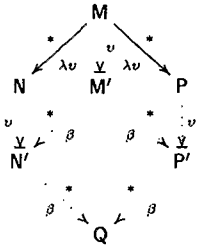
5. Un trait est un concept de LSL. C'est en fait un regroupement d'axiomes auquel on donne un nom.

6. Le lecteur curieux remarquera qu'on ne peut pas faire correspondre à Ω un programme CAML, car il faudrait que x dans $(\lambda x\ xx)$ soit à la fois une fonction de type a' ->b' et un objet de type a'. En fait, un théorème fondamental du lambda-calcul dit que les termes typables sont fortement normalisables. Dans les langages de programmation comme CAML, on ajoute aussi des définitions récursives qui ne font plus partie du lambda calcul classique (construction let rec), qui peuvent mettre en danger la terminaison, mais qui rendent CAML très puissant.

$M \xrightarrow[B]{} N$ alors la projection de M par υ se β-réduit vers la projection de N par υ:



De là on déduit la *confluence* du calcul $\lambda\upsilon$ sur les termes contenant des clôtures (termes clos) à partir du schéma suivant:



Nous avons vu que certains termes comme Ω ne sont pas fortement β-normalisables, c'est-à-dire qu'ils peuvent donner lieu à une suite infinie de β-réductions, mais on peut se demander si *un terme fortement β-normalisable est aussi fortement $\lambda\upsilon$-normalisable.* La réponse est oui [2], mais elle n'est pas si évidente car il existe des calculs de substitutions explicites qui ne satisfont pas cette propriété [14]. On dit que $\lambda\upsilon$ *préserve la normalisation forte* (propriété PSN).

7 La machine à triades

Nous venons de voir la réduction en forme normale qui peut-être réalisée par le calcul de substitutions explicites $\lambda\upsilon$. Cette forme normale est appelée *forme normale forte* en lambda calcul, car il existe une *forme normale faible* qui correspond plus à l'évaluation (donc au calcul) dans les langages de programmation fonctionnelle. La normalisation qui l'obtient s'interdit d'appliquer la règle (B) si elle apparaît sous un λ.

Les calculs de substitutions explicites offrent une large palette de stratégies de réduction à la forme normale faible des termes du lambda-calcul ; mais alors se posent les questions de savoir dans quel ordre on réduit les redex et comment on distribue les substitutions créées. La première question se pose en lambda calcul pur, tandis que la deuxième est typiquement du ressort des calculs de substitutions explicites. Les stratégies sont des réponses à des questions plus précises encore. *Évalue-t-on le corps de la fonction ou le paramètre en premier? Tente-on de réutiliser des normalisations effectuées auparavant en évitant de répéter les calculs déjà faits?* Si l'on évalue le corps de la fonction en premier, on parle d'*appel par nom*, si on évalue le paramètre en premier, on parle d'*appel par valeur*, si l'on évite de refaire une normalisation de sous-terme déjà faite, on parle d'*appel par nécessité* ou de *stratégie paresseuse*.

Pour abstraire la description d'une maquette d'implantation du lambda calcul et de la stratégie qui lui est associée on utilise une machine abstraite et plutôt que de décrire les différentes machines abstraites, nous allons utiliser une machine très générique que nous avons décrite dans [3], que nous avons appelée la *machine à triades* et qui peut-être instanciée en diverses machines. La présentation de la machine à triades nous donnera l'occasion de présenter diverses techniques d'implantation des langages fonctionnels et différentes machines abstraites.

Dans la machine à triades il y a trois concepts fondamentaux : la mémoire ou *tas*, les *transitions* de la machine (qui sont en fait ses *instructions*) et la *stratégie* qui dicte la façon d'invoquer les transitions.

machine à triades
=
tas + transitions + stratégie

$s \xrightarrow{*} s' \in \mathcal{T}$ si et seulement si :

$a \in Dom(s)$, et l'une des transitions suivantes peut-être appliquée :

(APP) : Si $s(a) = (MN, e, p)$ alors $\left\{ \begin{array}{l} s'(x) = \left\{ \begin{array}{ll} (M, e, b.p) & \text{si } x = a \\ (N, e, []) & \text{si } x = b \\ s(x) & \text{autrement.} \end{array} \right. \text{, où } b = \texttt{new}(Dom(s)) \\ \text{avec } Dom(s') = Dom(s) \cup \{b\}. \end{array} \right.$

(LMDA) : Si $s(a) = (\lambda M, e, b.p)$, alors $s'(x) = \left\{ \begin{array}{ll} (M, b.e, p) & \text{si } x = a \\ s(x) & \text{autrement.} \end{array} \right.$

(VARS) : Si $s(a) = (\underline{n+1}, b.e, p)$, alors $s'(x) = \left\{ \begin{array}{ll} (\underline{n}, e, p) & \text{si } x = a \\ s(x) & \text{autrement.} \end{array} \right.$

(VAR0) : Si $s(a) = (\underline{0}, b.e, p)$, alors $s'(x) = \left\{ \begin{array}{ll} (N, e', p' + p) & \text{si } x = a \\ s(x) & \text{autrement.} \end{array} \right.$ avec $(N, e', p') = s(b)$

où $+$ est la concaténation des listes.

FIG. 7 *Les transitions*

Nous allons décrire ces concepts dans l'ordre.

Le tas C'est un ensemble de petites entités de mémoire appelées *triades* et constituées de trois composantes le *code*, l'*environnement* et la *pile*. Chaque triade est repérée par une adresse. Le code est un terme avec indices de de Bruijn. L'environnement et la pile sont des listes d'adresses. Intuitivement l'environnement $a_0\, a_1 \ldots a_n$ contient en a_0 l'adresse de la triade où se trouve (à la place du code) le terme correspondant à l'indice $\underline{0}$ du code courant, en a_1 l'adresse de la triade où se trouve le terme correspondant à l'indice $\underline{1}$ et ainsi de suite jusqu'à n. À un instant donné, on peut supposer que l'on évalue le corps d'une fonction qui a p paramètres. La liste $b_0 \ldots b_{p-1}$ contient les adresses où retrouver ces paramètres que l'on suppose avoir «empilés». Le tas se trouve à un instant donné dans un état noté s ; c'est une fonction qui à une adresse a dans $\mathcal{A}$ fait correspondre une triade $s(a)$. Un état donné ne contient que des triades dans un sous-ensemble noté $Dom(s)$ de l'espace d'adresses $\mathcal{A}$; $Dom(s)$ est appelé le *domaine* de s.

Les transitions Elles décrivent les changements imposés au tas par le code qui se trouve dans une triade donnée choisie par la stratégie. Ces changements sont assez locaux, car ils n'affectent que la triade avec tout au plus la création d'une nouvelle triade (figure 7). $s \xrightarrow{*} s'$ est une modification du tas qui passe de l'état s à l'état s' ; cette modification est imposée par le contenu de la triade $s(a)$.

La stratégie La stratégie dit sur quelle triade doit s'effectuer la transition courante.

Charger un programme ou initialiser une machine avec un terme M correspond à créer un état s_0 du tas réduit à une seule triade localisée à l'adresse a_I ; la triade $s_0(a_I)$ contient le code M, un environnement vide et une pile vide.

8 Quelques stratégies

Nous allons examiner trois stratégies appelées respectivement *appel par nom*, *appel par valeur* et *appel par nécessité* ou *stratégie paresseuse* qui vont être illustrées par un exemple. Le lecteur qui désire une approche plus rigoureuse est invité à se rapporter à [3]. Les trois stratégies sont illustrées sur la réduction du terme $(\lambda\underline{0}\underline{0})\, I\, I$, où I et le terme $\lambda\underline{0}$ qui apparaît dans les figures 8 et 9

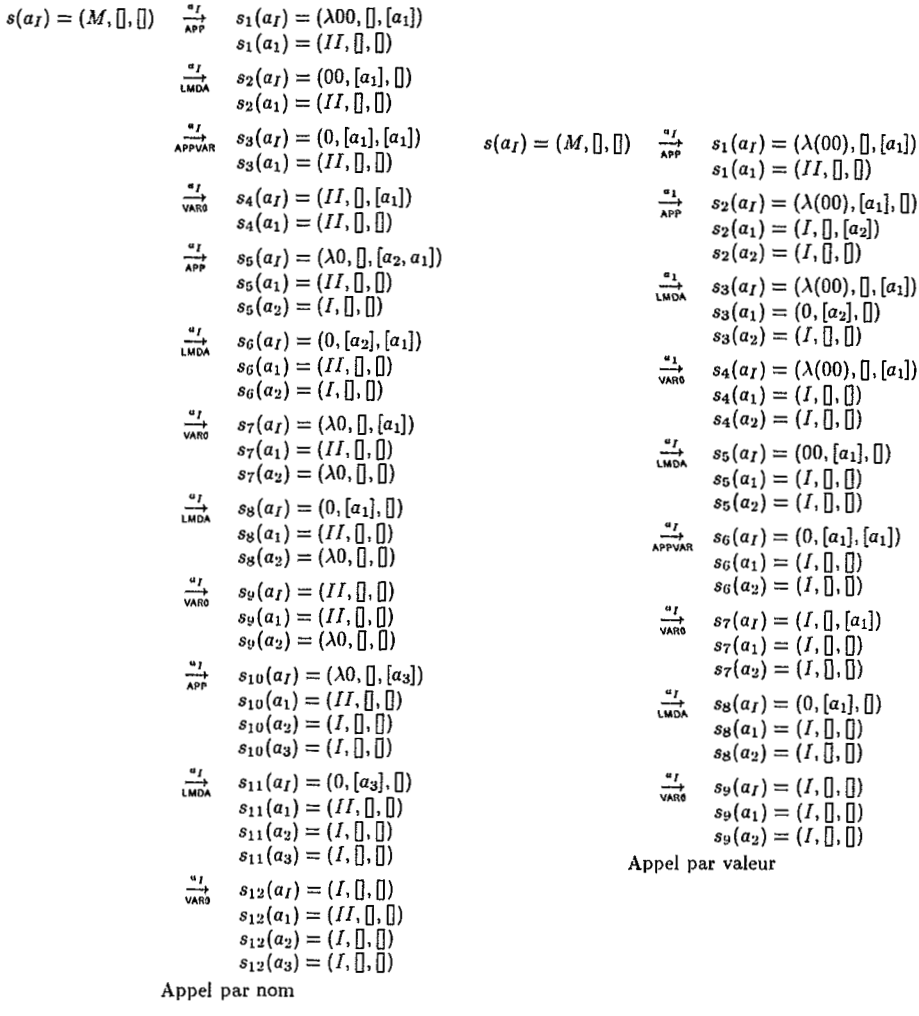


FIG. 8 *Évaluation dans la machine à triades du terme* $(\lambda\underline{00})I\,I$ *par appel par nom et par appel par valeur.*

$$
\begin{array}{lll}
s(a_I) = (M, [], []) & \xrightarrow[\text{APP}]{\bullet_I} & s_1(a_I) = (\lambda 00, [], [a_1]) \\
 & & s_1(a_1) = (II, [], []) \\
 & \xrightarrow[\text{LMDA}]{\bullet_I} & s_2(a_I) = (00, [a_1], []) \\
 & & s_2(a_1) = (II, [], []) \\
 & \xrightarrow[\text{APPVAR}]{\bullet_I} & s_3(a_I) = (0, [a_1], [a_1]) \\
 & & s_3(a_1) = (II, [], []) \\
 & \xrightarrow[\text{APP}]{\bullet_1} & s_4(a_I) = (0, [a_1], [a_1]) \\
 & & s_4(a_1) = (I, [], [a_2]) \\
 & & s_4(a_2) = (I, [], []) \\
 & \xrightarrow[\text{LMDA}]{\bullet_1} & s_5(a_I) = (0, [a_1], [a_1]) \\
 & & s_5(a_1) = (0, [a_2], []) \\
 & & s_5(a_2) = (I, [], []) \\
 & \xrightarrow[\text{VAR0}]{\bullet_I} & s_6(a_I) = (0, [a_1], [a_1]) \\
 & & s_6(a_1) = (I, [], []) \\
 & & s_6(a_2) = (I, [], []) \\
 & \xrightarrow[\text{VAR0}]{\bullet_I} & s_7(a_I) = (I, [], [a_1]) \\
 & & s_7(a_1) = (I, [], []) \\
 & & s_7(a_2) = (I, [], []) \\
 & \xrightarrow[\text{LMDA}]{\bullet_I} & s_8(a_I) = (0, [a_1], []) \\
 & & s_8(a_1) = (I, [], []) \\
 & & s_8(a_2) = (I, [], []) \\
 & \xrightarrow[\text{VAR0}]{\bullet_I} & s_9(a_I) = (I, [], []) \\
 & & s_9(a_1) = (I, [], []) \\
 & & s_9(a_2) = (I, [], [])
\end{array}
$$

FIG. 9 – *Évaluation dans la machine à triades du terme* $(\lambda\underline{00})I\,I$ *par appel par nécessité.*

Appel par nom Dans cette stratégie les transitions se font toujours par examen de la triade à l'adresse a_I. Les autres triades ne servent qu'à stocker des résultats intermédiaires.

Dans l'exemple, l'argument $(I\,I)$ est calculé deux fois parce qu'il est instancié deux fois, le premier calcul commence à l'état s_4 et finit à s_8 et le second commence à s_9 et finit à s_{12}.

Appel par valeur Dans le cas où une transition APP s'effectue sur une adresse a, l'appel par valeur transfère l'évaluation à la triade nouvellement créée jusqu'à ce que celle-ci atteigne sa forme normale, auquel cas elle rend le contrôle à la triade qui se trouve en a. Bien sûr, aucun transfert de contrôle n'a lieu si le paramètre est déjà en forme normale.

L'exemple montre que l'évaluation de l'argument $(I\,I)$ suit immédiatement la création de sa triade. Ensuite le calcul continue par l'évaluation du corps de la fonction.

Appel par nécessité ou stratégie paresseuse Au moment où l'on a besoin du terme associé à un indice, c'est-à-dire quand on veut appliquer la règle VAR0, on évalue d'abord ce terme dans sa triade d'origine avant de l'installer dans la triade appelante. Ce calcul resservira lors d'autres appels à la même valeur. La stratégie est dite «paresseuse» car elle évite de refaire des calculs inutilement.

L'évaluation de l'argument $(I\,I)$ est effectuée à la première instanciation et son résultat est naturellement stocké dans la triade associée, c'est-à-dire à l'adresse a_1. Pour la seconde instanciation, aucun calcul n'est nécessaire.

Dans l'exemple le nombre de transitions pour l'appel par nécessité est le même que pour l'appel par valeur mais le chemin d'exécution est différent, puisque la trace du calcul pour l'appel par nécessité est $a_I^3 a_1^3 a_I^3$ tandis que la trace du calcul pour l'appel par valeur est $a_I a_1^3 a_I^5$. Le nombre de pas de calculs est plus grand pour l'appel par nom. Il est possible de trouver des termes qui réclament plus de calcul pour l'appel par valeur que pour l'appel par nécessité. Par exemple, l'évaluation de $K\,I\,\Omega$ ne termine pas dans le cas de l'appel par valeur.

On peut démontrer que les règles sont correctes en montrant qu'elles simulent un calcul de substitutions explicites. De même, on peut prouver la correction des stratégies en les liant à des stratégies de réduction dans des calculs de substitutions explicites.

9 Conclusion

Cet article présente un aspect des recherches effectuées dans le projet EURÉCA du CRIN (CNRS) et de l'INRIA-Lorraine autour du thème des substitutions explicites. Ce thème d'informatique fondamentale englobe à la fois des aspects théoriques (autour du lambda calcul) et des aspects pragmatiques autour

de l'implantation des langages fonctionnels. Les recherches en cours portent notamment sur la résolution d'équations en lambda calcul (unification d'ordre supérieur) et l'implantation parallèle et distribuée des langages fonctionnels par la définition des nouvelles stratégies sur la machine à triade.

Références

[1] 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.

[2] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 1996. à paraître.

[3] Z. Benaissa and P. Lescanne. Triad machine, a general computational model for the description of abstract machines. unpublished, August 1995.

[4] N. Bourbaki. *Éléments de mathématiques: Théories des ensembles*, volume 1. Hermann & C^{ie}, 1954.

[5] A. Church and J. B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 40, 1936.

[6] H. B. Curry and Feys. *Combinatory Logic*, volume 1. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1958.

[7] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5):381 392, 1972.

[8] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[9] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*, chapter 12, pages 415 443. The MIT press, 1991.

[10] P. J. Landin. A correspondance between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8:89 101 and 158 165, 1965.

[11] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157 166, 1966.

[12] 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.

[13] P. Lescanne and J. Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 294 308. Springer-Verlag, 1995.

[14] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In M. Dezani, editor, *Int. Conf. on Typed Lambda Calculus and Applications*, 1995.

[15] K. Rose and R. Bloo. Deriving requirements for preservation of strong normalisation in lambda calculi with explicit substitution. Available as `ftp://ftp.diku.dk/diku/users/kris/Explicit-PSN.ps`, April 1995.

[16] P. Weis and X. Leroy. *Le langage Caml.* InterEditions, 1993.

[17] P. Weis and X. Leroy. *Manuel de référence du langage Caml.* InterEditions, 1993.