République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université M'hamed Bouguerra - Boumerdès

Faculté des Sciences
Département d'Informatique

Mémoire

Pour l'obtention du diplôme de

Magister en Informatique
Option : Informatique Fondamentale

Thème :

Une Machine Abstraite pour le système $\lambda c\beta^{+}$-calcul

---

JURY

- Mr. Mohamed Ahmed Nacer Maître de conférence Président de jury
- Mr. Ahmed Ouamar Maître de conférence Examinateur
- Mr. Mohamed Mezghiche Professeur Promoteur
- Mr. Djamel Zegour Maître de conférence Examinateur

Remerciements

Je tiens, tout d'abord, à remercier vivement le Pr Mohamed Mezghiche pour m'avoir proposé et dirigé mon mémoire. Ce travail n'aura pas pu voir le jour sans ses précieux conseils et ses lectures attentives de mes rapports.

J'adresse un grand merci à Mr. Mohamed Ahmed Nacer d'avoir accepté de présider mon jury.

Je remercie Mr. Ahmed Ouamar et Mr. Djamel Zegour pour m'avoir fait l'honneur d'examiner ce travail et d'avoir eu la gentillesse de faire partie de mon jury.

Aussi, je tiens à remercier Monsieur Baddari Kamel le Doyen de la faculté des Sciences et Monsieur Harzellah Abdelkrim le chef de département d'informatique pour leurs aides et leurs soutiens pendant toute la période de mes études au sein de la faculté des Sciences.

Introduction

Plusieurs années de recherches et de développement à améliorer et perfectionner les machines informatiques ont permis d'offrir à l'utilisateur des machines de plus en plus performantes à des prix de plus en plus réduits. Cependant, l'accroissement des puissances de calcul entretient une demande sans cesse accrue de puissance supplémentaire. Ceci pour des logiciels de plus en plus volumineux dont la mise au point et la maintenance deviennent problématiques. La maîtrise logicielle est loin d'égaler celle de la technologie. La production automatisée, voir même simplement la réutilisation de logiciel sont actuellement de véritable gageures. Ceci provient de la nature profonde des langages (impératifs) informatique utilisés. L'utilisation de langages développés à partir de concepts formels et non à partir de considération implémentatoires, devrait favoriser la formalisation et par la même l'automatisation de la programmation. Cependant, la généralisation de ce type de langages (Fonctionnels ou logiques) n'est pas aussi évidente pour deux raisons de bases. D'une part la chute des performances et d'autre part un savoir-faire logiciel traditionnellement imprégné du schéma de pensée « à la Von Neuman ».

Depuis l'apparition du premier langage fonctionnel lisp, plusieurs travaux ont été consacré pour la mise en œuvre de ce genre de langage. La première description formelle de mise en œuvre est représentée par la machine SECD[Lan 63]. Cette première étude fut à la base de toute une famille de mise en œuvres de langages fonctionnels : Les machines à environnement (CAM [CCM87], zinc[le90], la machine de krivine[kri85]….). En parallèle à cette famille de mise en œuvre une autre famille fut son apparition : Les machines à réduction de graphes. Les premiers réducteurs de graphe [wad71] utilisent un interpréteur. Une machine abstraite pour la réduction de graphe fut réalisée par Turner. Cette dernière n'a pas eu beaucoup de succès du fait que le code produit s'accroît de manière quadratique par rapport au code source**.** La G-machine [PJ90] est considérée pendant plusieurs années comme la représentative de cette famille.

La programmation fonctionnelle a pour base le λ-calcul, la réduction considérée par cette technique est la réduction faible de tête. La réduction forte par contre est exploitée par les systèmes de preuve et de démonstration automatique.

Une partie de recherches sur le λ-calcul se tourne actuellement vers des techniques de réductions efficaces des langages et des systèmes de preuves, parmi lesquelles certaines se tournent vers les systèmes de réécritures. Les λ-calcul avec substitution explicite, permettent d'utiliser la réécriture pour l'évaluation, le $\lambda C\beta^+$ calcul`[GMMS01]` est l'un de ces systèmes.

Ces systèmes peuvent être implantés sous la forme de machines à environnement ou de machine à réducteurs de graphes.

Ce mémoire est organisé en 3 chapitres. Le premier chapitre consiste en une introduction du lambda calcul. Le second chapitre est divisé en deux partie. La première partie est dévisée en deux sections. La première section est consacrée pour la présentation des méthodes d'implantation des langages fonctionnels. Pour chaque méthode, les machines de réduction de base sont présentées. On s'y intéressé surtout à mettre en relief la nouveauté que chaque machine apporte. Dans la seconde section, on présente les machines définies pour la réduction forte. Dans la deuxième partie du chapitre, en présente les calculs à substitution explicites. Le troisième et dernier chapitre présente la machine de réduction qu'on a défini implémentant l'algorithme de réécriture de terme $\lambda C\beta^+$ calcul. Des exemples détaillés sont donnés en fin du chapitre afin d'illustrer le fonctionnement de la $\lambda C\beta^+$ machine. La réalisation de la Machine est donnée en annexe.

Chapitre I

Introduction au λ-calcul

1. Introduction

Le lambda-calcul est un formalisme très simple, proposé dans les années trente par Alonzo Church[Chu41] pour servir d'alternative à la théorie des ensembles comme fondement des mathématiques. Bien que ce programme n'ait pas eu de succès a l'époque, le lambda-calcul est cependant devenu l'un des outils incontournables de l'informatique théorique. Le lambda-calcul est en particulier le modèle sémantique de base de tous les langages fonctionnels. Un programme d'un tel langage représente une fonction, qui peut manipuler d'autres fonctions. En outre un lambda terme est un objet mathématique dont il est facile d'extraire du sens. Ce chapitre survole la syntaxe du lambda calcul et certains aspects qui nous serons utiles par la suite.

2.1 Syntaxe du λ–calcul pur

2.1.1 Formation des termes

On introduit les symboles '(' , 'λ', ')', '.' Ainsi qu'un certain nombre de variables infinie dénombrable (lettres de l'alphabet) notées X. La formation des termes du lambda calcul (λ-termes)se fait selon les règles de la grammaire suivantes :

2.1.2 Syntaxe d'une λ-expression (en forme BNF)

Constante ::= 'c'| <constante> ''

Variable ::= v | <variable> ''

EXP ::= <constante>	constante prédéfinie
| <variable>	nom de variable
| '('<EXP><EXP>')'	application
| '(λ' <variable>. <EXP> ')'	lambda (λ)abstraction

Un terme de la forme (λ x. M) est dit abstraction ; l'abstraction de la variable x dans le terme M.

Un terme de la forme MN est dit application ; l'application du terme M au terme N, tel que N est considéré comme paramètre formel de la fonction M.

Convention

En général on n'écrit pas les parenthèses les plus à gauches et on utilise les convention de notation suivantes :

$\lambda x_1.x_2...x_n.M = \lambda x_1.\lambda x_2 \ldots \lambda x_n.M = (\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M)\ldots))$.

$FM_1..M_n = (\ldots(FM_1)...M_n)$

Remarque.

Dans cette définition, nous manipulons deux sortes distinctes de variables : les variables notées x,y, z, . . . qui sont les variables du *λ*-calcul, et les *metavariables M,N,P,* . . . qui servent à décrire les termes du *λ*-calcul, et sont donc définies à un niveau externe de description.

2.1.3. Relation d'occurrence

La relation d'occurrence est définie comme suit : soient P et Q deux λ-termes, par induction avec les clauses suivantes :

- P occure dans P (P est un sous-terme de P)
- Si P occure dans M ou dans N alors P occure dans le terme (M N)
- Si P occure dans M ou P ≡x alors P occure dans le terme (λx.M)

2.1.4. Champ d'un lambda terme

Pour une occurrence de terme (λx.M) dans un terme P, l'occurrence de M est appelée le champ de λ se trouvant à gauche.

2.2. Notion de Variable libre, variable liée

Une occurrence de variable est liée si elle est située à l'intérieur d'une lambda abstraction qui la lie, sinon elle est libre.

Notation :

Soit BV(M), l'ensemble des variables liées de M définie inductivement par :

$BV(x) = \varnothing$

$BV(\lambda x.M) = BV(M) \cup \{x\}$

$BV(MN) = BV(M) \cup BV(N)$

Soit FV(M), l'ensemble des variables libres de M définie inductivement par :

$FV\{x\} = \{x\}$

$FV(\lambda x.M) = FV(M) - \{x\}$

$FV(MN) = FV(M) \cup FV(N)$

Exemple1 Dans le terme $(\lambda x.(\lambda y.x\ y)\ z)$ l'occurrence de x et l'occurrence de y sont liées tandis que l'occurrence de z est libre.

Une occurrence de variable doit être ou bien libre ou alors liée. Une variable peut être liée et libre dans un même terme. On dit que la variable apparaît liée pour une occurrence et apparaît libre pour une occurrence.

Exemple2 Dans le terme $((\lambda x.(\lambda y.x\ y))\ x)$, la première occurrence de x apparaît liée ainsi que l'occurrence de y. La seconde occurrence de x apparaît libre.

Un terme qui ne contient aucune variable libre est appelé **terme clos** ou encore **combinateur**.

2.3 α-conversion

Elle permet de changer le nom des paramètres formels d'une lambda abstraction. Les règles suivantes donnent la définition formelle de la α-Conversion

- $\lambda x.N \equiv_\alpha (\lambda y.(N[y/x]))\ \ y \notin N$
- $$\frac{M_1 \equiv_\alpha N_1 \qquad M_2 \equiv_\alpha N_2}{M_1 M_2 \equiv_\alpha N_1 N_2}$$
- $$\frac{M \equiv_\alpha N}{\lambda x.M \equiv_\alpha \lambda x.N}$$
- $x \equiv_\alpha x$

L' α-conversion ne change pas la « signification » des termes.

2.4 Convention (Convention de noms de Barendregt)

Si $M_1, \ldots, M_n$ apparaissent dans un certain contexte mathématique (définition, preuve), alors dans ces termes, toutes les variables liées sont choisies telles qu'elles aient des noms différents.

Remarque :

La convention de nom implique un renommage implicite des variables lors des réductions. Par exemple, le terme (λx.xx)(λyz.yz) satisfait la convention de Barendregt. Si on le réduit machinalement, c'est-`a-dire sans respecter la convention en cours de réduction, on obtient la réduction suivante : (voir plus loin pour la définition de la β-réduction « $\rightarrow \beta$ »)

$(\lambda x.xx)(\lambda yz.yz) \rightarrow_\beta (\lambda yz.yz)(\lambda yz.yz) \rightarrow_\beta \lambda z.(\lambda yz.yz)z \rightarrow_\beta \lambda z.\lambda z.zz$

La *bonne* réduction est en fait la suivante :

$(\lambda x.xx)(\lambda yz.yz) \rightarrow_\beta (\lambda yz.yz)(\lambda y'z'.y'z') \rightarrow_\beta \lambda z.(\lambda y'z'.y'z')z \rightarrow_\beta \lambda z.\lambda z'.zz'$

2.5 Substitution

On appel substitution d'un terme à une variable, le remplacement de toute les occurrences libres de cette variable par le terme considéré. Ceci implique le remplacement des variables liées du terme considéré qui risque de lier les variables libres du terme en question. Plus formellement le mécanisme de substitution peut être décrit par le système de règle de la figure (fig1).

Cette opération de remplacement (avec l'application de l'axiome d'α conversion) est coûteuse du fait qu'elle doit parcourir récursivement le terme à substituer. [DOU96]

Dans la pratique, les mises en oeuvres retardent la propagation de la substitution jusqu'à ce qu'elle soit nécessaire.

$$
\begin{aligned}
x[N/x] &\equiv N \\
a[N/x] &\equiv a \text{ avec a variable différente de x} \\
(\lambda x.y)[N/x] &\equiv (\lambda x.y) \\
(\lambda y.Y)[N/x] &\equiv (\lambda y.(Y[N/x])) \text{ si } (y \notin FV(N) \text{ et } (x \neq y)) \text{ ou } x \notin FV(Y) \\
&\equiv \lambda z.(Y[z/y][N/x]) \text{ si } (y \in FV(N)) \cap (x \in FV(Y)) \\
(M_1 M_2)[N/x] &\equiv (M_1[N/x])(M_2[N/x])
\end{aligned}
$$

Fig 1 :Définition formelle de la substitution

2.6. β-réduction

Un terme de la forme ($\lambda\, x.M)N$ est appelé *redex* (ou encore radical). La β-réduction, c'est la réduction d'un redex à l'intérieur d'un terme. Plus précisément, c'est la relation définie par la figure (fig2)

$(\lambda x.M)N \rightarrow$	$M[N/x]$.	β
Si $M \rightarrow_\beta N$	alors $\lambda\, x.M \rightarrow_\beta \lambda\, x.N$.	ξ
Si $M_1 \rightarrow_\beta N$	alors $M_1 M_2 \rightarrow_\beta N M_2$.	ν
Si $M_2 \rightarrow_\beta$ N	alors $M_1\, M_2 \rightarrow_\beta M_1 N$.	μ

Fig2 :définition formelle de la β-réduction

Si l'on considère le terme $((\lambda x.M)N)$, la β-réduction peut être comprise comme le remplacement du paramètre formel x, ayant servi à décrire la fonction (λx.M) par le paramètre effectif N. La β-réduction est l'opération de base implantée par les mises en œuvres de langages fonctionnels.

2.6.1 βη-réduction, βη-équivalence :

Outre le remplacement des arguments formels par les arguments réels, une autre transformation comparable des termes peut être incorporée à l'équivalence. C'est la transformation de λx.(t x) en t quand x n'occure pas dans t. Cette transformation permet par exemple d'identifier les termes M et λx.($M\, x$).

Un $\beta\eta$-radical est un terme de la forme $\lambda x.(t\, x)$, où la variable x n'apparaît pas dans le terme t.

La relation entre termes $t \rightarrow_{\beta\,\eta} u$ (t se $\beta\eta$ réduit en une étape en u) est la plus petite relation telle que :

- $((\lambda x\ .\ M)\ N) \rightarrow_{\beta\,\eta} M[N/x]$,
- $\lambda x\ .(M\ x) \rightarrow_{\beta\,\eta} M$ *si* x *n'apparaît* pas dans M,
- *si* $M \rightarrow_{\beta\,\eta} N$ *alors* $(M\ M_2) \rightarrow_{\beta\,\eta} (N\ M_2)$,
- *si* $M \rightarrow_{\beta\,\eta} N$ *alors* $(M_1\ M) \rightarrow_{\beta\,\eta} (M_1\ N)$,
- *si* $M \rightarrow_{\beta\,\eta} N$ *alors* $\lambda x\ .\ M \rightarrow_{\beta\,\eta} \lambda x.N$.

On définit de même les relations $\rightarrow^*_{\beta\,\eta}$ et $\equiv_\beta$ comme les fermetures réflexive-transitive et réflexive-symétrique-transitive de cette relation.

On note également $t \to_\eta u$ (t se η-réduit en une étape en u), la plus petite relation telle que :

- $\lambda x.(M\ x) \to_\eta MN$ si x n'apparaît pas dans t,
- si $M \to_\eta N$ alors $(M\ M_2) \to_\eta (N\ M_2)$,
- si $M \to_\eta N$ alors $(M_1\ t) \to_\eta (M_1 u)$,
- si $M \to_\eta N$ alors $\lambda x.M \to_\eta \lambda x.N$.

2.7 Forme normale

Lorsqu'une expression E ne possède aucun radical alors, elle est dite sous forme normale. Certain termes n'ont pas de forme normale .

Exemple

$(\lambda x.xx)y \to_\beta yy$

$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta \ldots$

La seconde expression se réduit en elle même; elle est indéfiniment réductible .

2.7.1 Terme normalisable

On dit qu'un terme M est normalisable s'il existe une β-réduction $M \to^{\beta}_{*} N$ (M se réduit en plusieurs étapes en N) où N est en βforme normale.

2.7.2 Terme faiblement normalisant

On dit qu'un terme M est faiblement normalisant s'il existe au moins une β-réduction $M \to^{\beta}_{*} N$ où N est en βforme normale.

2.7.3 Terme fortement normalisant

Un terme *u* est fortement normalisant si et seulement si toutes les réductions partant de *u* sont finies.

2.7.4 Faible forme normale de tête

Un terme μ est sous forme normale de tête s'il n'a pas de redex de tête. Un redex de tête d'un terme est un redex qui n'est pas à droite d'une application. Toute expression de la forme $F\ E_1E_2\ ..\ E_n$ est sous forme normale de tête, avec $(F\ E_1E_2\ ..\ E_n)$ n'est un radical pour aucun $m \leq n$.

2.8 Ordre de réduction

Une expression peut contenir plus d'un radical dans ce cas la, la réduction peut se faire selon plusieurs chemins.

Exemple

$$(\lambda x.x)((\lambda x.y)\ (\lambda x.x)) \xrightarrow{A} ((\lambda x.y)\ (\lambda x.x)) \longrightarrow y$$

$$(\lambda x.x)((\lambda x.y)\ (\lambda x.x)) \xrightarrow{B} (\lambda x.x)\ (y) \longrightarrow y$$

Le problème qui se pose maintenant, est de savoir si deux chemins de réduction différent peuvent aboutir à des formes normales différentes. Les théorèmes de church-Rosser(**A** et **B**) répondent négativement à la question.

2.8.1 Théorème de church-Rosser(A)

Si $E_1 \leftrightarrow E_2$, alors il existe une expression E telle que :

$E_1 \rightarrow E$ et $E_2 \rightarrow E$

2.8.2 Théorème II de Church-Rosser(B)

Si $E_1 \rightarrow E_2$ et si E_2 est sous forme normale alors il existe une suite de réduction de E_1 vers E_2 suivant l'ordre normal.

2.8.3 Ordre Normal

La réduction en ordre normal spécifie que le radical le plus extérieur et le plus à gauche doit être réduit en premier. Dans l'exemple précédent la réduction par *le chemin A* est l'ordre normal de réduction .

2.8.4 Théorème (Barendregt)

La réduction à gauche est normalisante c'est à dire se termine si et seulement si le terme est normalisable.

2.8.5 Ordre optimal

Le problème de trouver des ordres de réduction pratiquement optimaux préservant les bonnes propriétés de l'ordre normal a été traité dans [lévy 80], il a été montré que dans la réduction de graphe, l'ordre normal de réduction est optimal.

Chapitre II

Mises en œuvres de langages fonctionnels

2.1. Model d'implantation des langages fonctionnel

Il existe une grande diversité de langages fonctionnels. Mais tous ont un point commun. La β-réduction, l'opération de base qui permet de calculer leurs résultats. En effet, il a été montré que tout langage fonctionnel peut être traduit en lambda calcul. Dans ce cas, la réduction d'une lambda expression simule alors l'exécution d'un programme fonctionnel source. Le schéma de la figure montre les étapes d'implantation d'un langage fonctionnel.

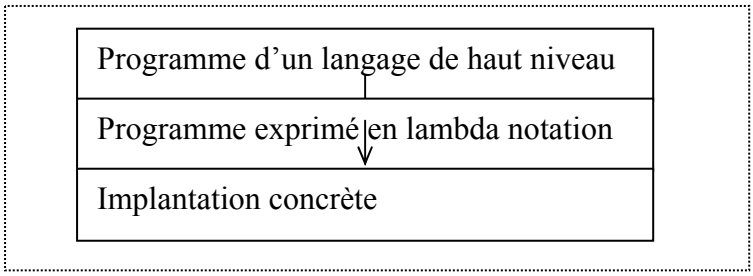


Fig3 : Implantation d'un langage fonctionnel

Dans la première étape chaque construction du langage de haut niveau doit être traduite vers des constructions du lambda calcul, en seconde étape, le code résultat (lambda calcul) doit exécuté. Pour ceci plusieurs façons sont possibles entre autre par l'implantation d'une machine virtuelle (abstraite) qui se chargera de l'exécution. Dans ce qui suit en fera un survole des machines existantes, comme il en existe une grande diversité en va se limiter au plus importantes.

2.1.1 Les stratégies d'évaluations

Un élément fondamental de la puissance d'expression des langages fonctionnels est la notion de fonction non stricte. Une fonction est stricte en un de ces arguments lorsque la valeur de

cet argument est nécessaire à l'évaluation du corps de la fonction ; dans le cas contraire, la fonction est dite non stricte pour cet argument. L'exemple le plus classique est la fonction (If A then B else C), l'évaluation de cette fonction requiert l'évaluation de A (le test) ; par contre on ne peut prédire lequel de B ou C sera ensuite évalué. On dit que la fonction If est stricte en son premier argument et non stricte en son $2^{ème}$ et $3^{ème}$ argument.

a. L'évaluation par valeur

Les langages stricts utilisent un mode d'évaluation par valeur. C'est à dire que l'argument d'une fonction est normalisé avant d'être passé en paramètre. Le passage des paramètres aux fonctions est dit dans ce cas par valeur (eval-apply). Les premières machines de réduction tel que SECD[LAN64], CAM[CCM87], Krivine[Kri85]…ETC, ont adopté cette stratégie.

b. L'évaluation par nom

Les arguments ne sont pas calculés au passage de paramètres. L'évaluation effective n'est faite que lorsque la machine accède par le biais d'une variable à la valeur de l'argument. Le passage des paramètres aux fonctions est dit par nom (push-enter). Bien que ne soient évalués que les arguments réellement utilisés, ce mécanisme est très coûteux puisqu'il nécessite de recalculer à chaque accès la valeur de l'argument. Des résultats théoriques montrent que l'évaluation par nom permet une définition maximale des résultats. C'est à dire que si une expression est calculable, son évaluation par nom nous donne toujours son résultat, ce qui n'est pas le cas de l'évaluation par valeur.

c. L'évaluation paresseuse

L'argument est évalué la première fois que sa valeur est nécessaire pour l'évaluation du corps de la fonction. Ceci correspond au mode de passage par besoin des langages impératifs. Comme on l'a vue dans la fonction "if", seuls deux arguments sur trois sont évalués. Dans certains langages fonctionnels, l'évaluation paresseuse est définie implicitement. En clair, l'exécution du programme n'évaluera les termes que lorsqu'ils seront indispensables à la poursuite du calcul. Ce qui fait que l'argument peut être évalué zéro ou une fois; il n'est en particulier pas évalué s'il n'est pas référencé lors de l'évaluation de la définition de la fonction.

Remarque1 : Les langages implantant des mécanismes paresseux tel que Haskell, Lazy ML, etc. utilisent en fait une forme légèrement différente de cette évaluation dite évaluation par

nécessité. Les valeurs ne sont calculées qu'une seule fois au premier accès et sont ensuite marquées.

Remarque2 : la notion de forme normale n'est pas la même pour une machine stricte et une machine paresseuse. En effet, une cloture d'une machine stricte ne contient que des valeurs en forme normale puisque la machine les évalue avant de les stocker. Ceci n'est pas le cas pour une machine paresseuse puisque celle-ci ne calcule (donc ne normalise) les valeurs de son environnement que lorsqu'elle y accède.

2.2. Les approches d'implémentation de langage fonctionnel

L'implémentation de langage fonctionnel est devenue un art en lui-même. Il existe plusieurs approches :

2.2.1 Approche par environnement

2.2.1.1 La machine SECD

La machine SECD fut la première machine proposée pour l'implémentation de langage fonctionnel par [Lan 63]. La machine SECD a été conçue pour décrire la sémantique opérationnelle des langages fonctionnels. Cette dernière est fondée sur la notion d'environnement (liste de pair variable + valeur avec une mise à jour dynamique). Elle met en œuvre l'appel par valeur, commence par le sous-terme le plus à droite. Elle possède quatre registres globaux :

1- S : La Pile qui contient les résultats intermédiaires durant l'évaluation,
2- E : Environnement, contient les valeurs associées aux variables,
3- C : le Code, une structure dont l'élément directement accessible correspond à l'expression évaluée ou à l'instruction exécutée.
4- D (Dump ou dépôt) : Zone de stockage d'autres registres utilisée quand la machine exécute une nouvelle procédure, (précédent état). Ceci correspond à peu prés à une pile de retour.

La machine SECD peut être vue comme une machine à transition d'état (un automate). Ces règles de transitions sont comme suit :

Avant de présenter ces règles on donne la signification des notations qu'on va utiliser. <S,C, E,D> représente un quadruplet où

- **s** désigne la pile où seront stockés les arguments en cours de réductions ;
- **e** l'environnement, contient les valeurs des variables (variable + valeurs) ;
- **c** le code, les suites d'instructions à réduires ;
- **d** le dûmp, la pile qui sauvegarde l'état précédent.

Une pile dont le sommet est n est notée n : S où S est le reste de la pile. Une pile vide est notée [].

Une séquence de code dont la première instruction est ***I*** est notée ***I :C*** où ***C*** est le reste du code. Une séquence de code vide est notée [].

Une sauvegarde est notée ***(S,E,C,D).*** La sauvegarde vide est notée ([],[],[],[]).

a- Les états de transitions de la machine SECD

Les états de transitions sont définis par type d'expression. Pour chaque type, il existe un état de transition qui correspond. Les types d'instruction possible sont :

const pour les constantes;

var pour les variable ;

app pour l'application ;

clos pour les clotures.

Remarque : Les constantes comprennent : Les entiers, les booléens, les opérateurs arithmétique et logique, les fonctions telles que (+,* …).

1. <S,E,k:C,D> $\longrightarrow_{\textbf{Const}}$ <k:S,E,C,D> ; l'instruction à réduire est la constante k, qui serai empilée sur la pile **s**.
2. <S,E,x:C,D> $\longrightarrow_{\textbf{Var}}$ <a:S,E,C,D> ; l'instruction à réduire est la variable x, sa valeur (soit a) va être recherché dans l'environnement E et puis mise sur le sommet de pile S.
3. <S,E,λx.M:C,D> $\longrightarrow_{\textbf{Clos}}$ <clos(x,M,E):S,E,C,D> ; l'instruction à réduire est une abstraction (λx.M). L'abstraction va être traduite sous forme de cloture et puis mise sur le sommet de la pile S.
4. <S,E,MN :C,D> $\longrightarrow_{\textbf{APP}}$ <s,e,N :M :app,c,d>; l'instruction à réduire est une application (MN), Cette expression va se traduire par la liste de l'expression N, l'expression M puis l'expression app.
5. <clos(x,M,e'):a:S,E,app:C,D> $\longrightarrow_{\textbf{appl}}$ <[],x=a:E',M,(S,E,C,D)>; l'instruction à réduire est **app** et au sommet de la pile il y a l'instruction cloture. Un lien est crée entre la variable(x) figurant dans la cloture et la valeur (a) qui doit être sur la pile (l 'élément suivant sur la pile), L'environnement va être enrichi par la variable et sa valeur, le corps de l'abstraction (M) va être mis dans le code, l'état actuel (S,E,C,D)va être sauvé dans la pile dump D.
6. < f:a:S,E,app:C,D)> $\longrightarrow_{\textbf{apply}}$ < f(a):S,E,C,D>; l'instruction à réduire est **app** et au sommet de la pile il y a une fonction primitive. Ceci va se traduire par un appel à la fonction en sommet de la pile avec comme argument l'élément suivant sur la pile. L'application de la fonction à l'argument est immédiat et le résultat est sur la pile S.
7. <a, E',[],(S,E,C,D)> $\longrightarrow_{\textbf{return}}$ <a:S,E,C,D>; Le code est vide, la pile d contient encore des valeur. Il s'agit donc de restaurer l'état précédent.
8. <a, [],[],[]> a, est donné en résultat, la machine s'arrête quand c et d sont vides.

Exemple

Voyions comment se fait la réduction du terme *twice sqr 3*, avec *twice =λfx.f(fx)*

Au départ, la pile S, l'environnement E ainsi que la pile D sont vides.

<[],[],twice sqr 3,[]> $\xrightarrow{\textbf{App}}$ <[],[],3:twice sqr: app,[]>

<[],[],3:twice sqr: app,[]> $\xrightarrow{\textbf{const}}$<3,[], twice sqr: app,[]>

<3,[], twice sqr: app,[]> $\xrightarrow{\textbf{App}}$<3,[], sqr :twice: app : app,[] >

<3,[], sqr :twice: app : app,[] > $\xrightarrow{\textbf{const}}$ <sqr:3,[], twice: app : app,[] >

< sqr :3, [], twice: app : app, [] > $\xrightarrow{\textbf{clos}}$ <clos(f, λx.f(fx),[]): sqr :3, [], app : app, [] >

<**clos**(f, λx.f(fx), []): sqr :3,[], app : app,[] > $\xrightarrow{\textbf{appl}}$ <[] , f=sqr, λx.f(fx),(3,[],app,[]) >

<[], f=sqr , λx.f(fx),(3,[],app,[]) > $\xrightarrow{\textbf{clos}}$<clos(x,f(fx), f=sqr), f=sqr ,[], (3,[], app,[])>

<clos(x, f(fx),f=sqr),f=sqr, [] ,(3,[], app,[]) > $\xrightarrow{\textbf{return}}$< clos(x, f(fx),f=sqr):3, [] ,app,[]>

<clos(x, f(fx),f=sqr):3,[],app,[]> $\xrightarrow{\textbf{appl}}$ < [],x=3:f=sqr, f(fx),([],[],[],[])>

< [],x=3:f=sqr, f(fx),([],[],[],[])> $\xrightarrow{\textbf{app}}$ < [],x=3:f=sqr, (fx):f:app, ([],[],[],[])>

< [],x=3:f=sqr, (fx):f:app, ([],[],[],[])> $\xrightarrow{\textbf{app}}$< [],x=3:f=sqr, x:f: app:f: app, ([],[],[],[])>

< [],x=3:f=sqr, x:f: app:f: app, ([],[],[],[])> $\xrightarrow{\textbf{var}}$ < 3,x=3:f=sqr,f:app:f:app, ([],[],[],[])>

< 3,x=3:f=sqr,f:app:f:app, ([],[],[],[])> $\xrightarrow{\textbf{var}}$ < sqr:3,x=3:f=sqr,app:f:app, ([],[],[],[])>

< sqr:3,x=3:f=sqr,app:f:app, ([],[],[],[])> $\xrightarrow{\textbf{apply}}$ < 9,x=3:f=sqr,f:app,[]>

< 9,x=3:f=sqr,f:app, ([],[],[],[])> $\xrightarrow{\textbf{var}}$< sqr:9,x=3:f=sqr,app, ([],[],[],[])>

< sqr:9,x=3:f=sqr,app, ([],[],[],[])> $\xrightarrow{\textbf{apply}}$ < 81,x=3:f=sqr,[],([],[],[],[])>

< 81,x=3:f=sqr,[],([],[],[],[])>$\xrightarrow{\textbf{return}}$ < 81,[],[],[]>

La machine s'arrête donc sur le résultat de la réduction (le code C et le dump d sont vides)qui est sur le sommet de la pile.

Cette implémentation ne sait pas traiter les variables libres, tous les termes utilisés doivent être clos. Un autre inconvénient consiste en la réduction toujours de l'argument même si ce dernier ne serais jamais utilisé. Afin de contourner ce problème une seconde machine fut réalisée, la Lazy SEDC, l'idée est d'empiler l'argument sans qu'il soit réduit et de mettre à jour l'environnement avec la valeur de l'argument, la première fois que ce dernier soit réduit. Mais cette nouvelle version, selon David Turner rend le temps d'exécution dix fois plus long et apporte moins d'amélioration [Pau02].

2.2.1.2 La machine CAM

La CAM (Categorical Abstract Machine) définie dans [CCM87] est issue des travaux sur les logiques combinatoires catégoriques, qui sont des systèmes formels basés sur la théorie des catégories et semblables à la logique combinatoire.

Le lambda calcul considéré est légèrement différent du lambda calcul classique, c'est au fait le lambda calcul avec couple (association de couple valeur +environnement) .

On va rappeler les définitions et les propriétés relatives à la Logique Combinatoire Catégorique C.C.L. On va d'abord définir l'algèbre des termes de la Logique Combinatoire Catégorique.(Les définitions sont tirées du [HA87])

Définition1

La logique Combinatoire Catégorique pure, C.C.L, est l'algèbre libre bâtie sur un ensemble *Var* de variables avec la signature suivante :

1.*Id, Fst, Snd, App,* opérateurs d'arité 0 appelés respectivement Identité, première projection, seconde projection et Applicateur.

2. Λ, opérateur d'arité 1, appelé curryfication

3. °, opérateur d'arité 2, appelé composition

4.<,>, opérateur d'arité 2, qui représente la formation de la paire

Définition2 La Logique Combinatoire Catégorique Forte, *CCL$\beta\eta$SP*, est la théorie définie par les règles suivante :

(Ass)	(x°y)°z=x°(y°z)
(IdL)	Id°x = x
(IdR)	x°Id = x
(Fst)	Fst°<x, y> = x
(Snd)	Snd °<x, y> = y
(Dpair)	<x, y>°z = <x°z, y°z>
(FSI)	<Fst, Snd> = Id
(SP)	<Fst°x, Snd°x> = x
(DΛ)	Λ(x)°y = Λ(x°<y°Fst, Snd>)
(Beta)	App°<Λ(x),y> = x°<Id, y>
(AI)	Λ(App) = Id
(SΛ)	Λ(App°<x°Fst,Snd>) = x

Fig4 Les règles de la Logique Combinatoire Catégorique forte

On note que les règles (SP) et (SΛ) sont des conséquences des précédentes.

La machine CAM est fondée sur une interprétation opérationnelle des combinateurs catégoriques. La CAM a servi de base pour l'implantation du langage fonctionnel CAML. Cette machine met en œuvre l'appel par valeur gauche-droite et utilise des environnements chaînés. Les listes de code sont concaténées dynamiquement. Cette machine contrairement à la machine SECD n'utilise qu'une seule pile qui contiendra les clotures, environnement et adresse de retour.

L'état de la CAM est constitué d'une séquence de code, d'une pile et d'un environnement courant.

a. Les instructions de la machine CAM

Chaque combinateur correspond à une instruction de la machine CAM, sauf la paire[1] qui se décomposera en plusieurs instructions.

L'ensemble des instructions **Ins** de la machine est défini par :

Ins::=

Ins_F (instructions associées aux opérateurs de base),

Push : met l'accumulateur sur le sommet de pile,

Swap : échange le sommet de pile et l'accumulateur,

Cons : chaîne le sommet de pile à l'accumulateur,

Fst : prend le premier élément de l'accumulateur,

Snd : prend le deuxième élément de l'accumulateur,

Id : NOP

App : exécute le code de la cloture du 1er élément de l'accumulateur

Cur : fabrique une cloture (code + environnement)

Quote Val : n'évalue pas Val
Val peut prendre les valeurs :

Val::=
C : constante de base
| Val : liste d'instructions (cloture)
(Val,Val) : couple de valeurs

[1] La paire en fait n'appartient pas au lambda calcul pur mais au lambda calcul avec couple. Ce dernier a été introduit afin de pouvoir faire la traduction des lambda termes dans C.C.L.

b. Traduction du λ-calcul en CAM

Les λ-termes sont représentés sous forme de combinateurs catégorique, il faut donc définir un schéma de compilation qui se chargera de traduire les λ-termes en combinateurs catégoriques.

c. Schéma de compilation

Le schéma de compilation utilise un motif *p* pour mémoriser la place des variables.

$x(p,x)$	Snd
$x(x,p)$	Fst
$x(p_1,p_2)$	$(Snd;xp_2)$? $(Fst;xp_1)$
MNp	$<Mp,Np>; App$
$(M,N)p$	$<Mp,Np>$
$\lambda\ v.Mp$	$Cur(M(p,v)$

Tab1 schéma de compilation

La notation e_1 ? e_2 signifie la valeur de e_1 si elle existe, sinon celle de e_2. Le combinateur paire donne la suite d'instructions suivantes :

$<M,N> \rightarrow Push;M;Swap;N;Cons$

Exemple

Soit à réduire le λ-terme $(\lambda\ x.xx)(\ \lambda\ x.x)$,

Le schéma de compilation est donné par le tableau Tab1 :

Remarque : Les règles appliquées sont celles données ci-dessus avec Cur pour représenter l'opérateur de curryfication (Λ) et le « ; » pour l'opérateur de composition.

$[[(\lambda\ x.xx)(\ \lambda\ x.x)]]$	$<Cur(<Snd,Snd>;App),Cur(Snd)>;App$
$(Beta)$	$<Id,Cur(Snd)>;(<Snd,Snd>;App)$
$(Ass)(Dpair)(Snd)$	$<Cur(Snd),\ Cur(Snd)>;App$
$(Beta)$	$<Id,Cur(Snd)>;\ Snd$
(Snd)	$Cur(Snd)$
$[[\lambda\ x.x\ ]]$	$Cur(Snd)$

Tab2 schéma de compilation du λ-terme $(\lambda\ x.xx)(\ \lambda\ x.x)$

Le code CAM correspondant à ($\lambda\, x.xx$)($\lambda\, x.x$) est le suivant :

Push;*Cur*(*M*);*Swap*;*Cur*(*N*);*App*

où *M*=*xx* (<*Snd*,*Snd*> ;*App*) correspondant au code

Push;*Snd*;*Swap*;*Snd*;*Cons*;*App*

et *N*=*x* (*Snd*) au code *Snd* .

L'exécution du code montre les différents états de l'environnement, du compteur ordinal (code) et de la pile.

Accumulateur	*Code*	*Pile*
()	*Push*;*Cur*(*M*);*Swap*;*Cur*(*N*);*Cons*;*App*	[]
()	*Cur*(*M*);*Swap*;*Cur*(*N*);*Cons*;*App*	[()]
():M)	*Swap*;*Cur*(*N*);*Cons*;*App*	[()]
()	*Cur*(*N*) ;*Cons* ;*App*	[() :M)]
() :N)	*Cons* ;*App*	[():M)]
(():M), ():N))	*App*	[]
(),():N))	*M*	[]
(),():N))	*Push*;Snd;*Swap*;Snd;*Cons*;*App*	[]
(),():N))	Snd;*Swap*;Snd;*Cons*;*App*	[(),():N))]
():N)	*Swap*;Snd;*Cons*;*App*	[(),():N))]
(),():N))	Snd;*Cons*;*App*	[():N]
():N)	*Cons*;*App*	[():N)]
(():N),():N))	*App*	[]
(),():N))	*N*	[]
(),():N))	*Snd*	[]
():N)		[]

Tab 3 Schéma d'exécution du λ-terme*($\lambda\, x.xx$)($\lambda\, x.x$)*

La machine s'arrête quand soit le compteur ordinal est vide (comme dans cet exemple), soit une instruction manipule le sommet de pile et celle-ci est vide. La valeur de l'évaluation est alors contenue dans le registre environnement. Celle-ci correspond bien au résultat escompté (*N*).

c. Représentation des environnements

Les valeurs de l'environnement sont chaînées sous forme de liste. L'accès se fait par le numéro de la variable à rechercher.

2.2.1.3 La machine de Krivine

La machine de Krivine fut proposée par Jean-Luis Krivin[Kri85]. C'est une machine abstraite pour l'évaluation de l'appel par nom (évaluation paresseuse) du lambda calcul pur, c'est au fait un interprète simple du Lambda calcul utilisant la notation de **De Bruijn**[DB72] pour l'accès au variables. Cette machine ne réduit que des termes clos et s'arrête à la forme normale de tête faible (i.e. On ne réduit jamais à l'intérieur d'un lambda terme). Elle possède trois instructions, correspondant au trois constructions du lambda calcul : accès au variable, abstraction et l'application. La pile d'argument et aussi la pile utilisée pour les résultats. On travaille donc sur des états constitués d'un code (terme pur du lambda calcul), un environnement et la pile. L'environnement et la pile sont des listes de clotures, qui sont des couples <code, environnement>. L'environnement représente les substitutions à opérer suites à des β-réductions, 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.

a. Les instructions de la machine de Krivine

Cette machine est donc décrite par les trois règles suivantes.

instruction ::=

Access(n)
| Push(code ou label)
|Grab

1. Access(n) : Cette instruction permet d'accéder à la $n^{ème}$ valeur de l'environnement courant. Cette valeur, qui est une cloture, devient le code courant.

2. Push : Cette instruction crée à partir du code et de l'environnement courant, une nouvelle cloture qui est empilée dans la pile. Push sépare en deux blocs de code distincts, le code correspondant à une fonction et le code de son argument. L'environnement courant est dupliqué pour que fonction et argument en reçoivent une copie.

3. Grab : Cette instruction extrait un bloc de code de l'état pour enrichir l'environnement courant. C'est cette instruction qui opère la β-réduction. L'argument d'une fonction vient enrichir l'environnement d'une cloture. Si l'état ne comporte pas de second bloc de code, la transition n'est pas définie et la machine s'arrête. Ce cas correspond à une fonction ne

recevant pas son argument; dans un langage fonctionnel classique il est de coutume que l'exécution s'arrête.

b. Schéma de compilation

On note $[M]_\rho$ la compilation du terme *M* dans l'environnement ρ.

- $[MN]_\rho$ = Push l; $[M]_\rho$; Label l; $[N]_\rho$
- $[\lambda\, x.M]_\rho$ = Grab; $[M]_{\rho,x}$
- $[x]_\rho$ = Acces n (où n est la position de *x* dans ρ)

c. Schéma d'évaluation

Une fois les λ-termes traduits en instructions machines (abstraite), il est nécessaire d'avoir un interprète de cette machine. On donne ci dessous le schéma d'évaluation des instructions de la machine

Env	Code	Pile	env	Code	pile
e	Push c';c	s	*e*	c	(*c'*,*e*).*s*
e	Grab; c	$(c_0,e_0).s$	$(c_0,e_0).e$	c	*s*
e	(*)Grab;c	()	e	Arrêt	()
$(c_0,e_0).e$	Acc(n+1);c	*s*	*e*	Acc(n) ;c	*s*
$(c_0,e_0).e$	*Acc(0);c*	*s*	e_0	c_0	*s*

Le critère d'arrêt :

L'instruction *Grab* correspond à une abstraction. S'il n'y a pas d'argument sur la pile, cette abstraction ne trouve donc pas d'argument et le calcul s'arrête. Il n'y a pas de réduction sous le λ.

La stratégie d'évaluation est l'appel par nécessité, l'argument n'est évalué que lorsqu'on a besoin. De ce fait à chaque fois qu'il y a nécessité, l'argument est réévalué autant de fois. Ce qui n'est pas efficace.

2.2.4. Machine de krivine avec marque

Afin de permettre l'évaluation stricte, il faudrait avoir un mécanisme qui puisse permettre de réduire les sous termes d'un terme donné en leurs formes normales faibles. Le problème avec la machine de krivine est qu'elle ne s'arrête (arrête la réduction) que si sa pile est vide. Ce qu'il faut c'est un moyen qui permet de stopper la réduction même s'il y a encore des termes (arguments) dans la pile. Une façon de faire c'est d'utiliser une marque, ces marques seront incorporées avec les clôtures empilées. Les clôtures marquées ne seront pas mises dans l'environnement, la réduction peut continuer avec d'autre sous termes ou termes et la réduction courante est stoppée.

a. Présentation de la machine

Cette machine est présentée dans [Le96]. Elle est munie de quatre instructions, en plus des trois connues, *réduire (c)*, pour forcer la réduction de *c* à sa faible forme normale, l'instruction *grab* a une autre sémantique. Les clôtures marquées seront représentées par <c,e> au lieu de (c, e).

Env	Code	Pile	Env	Code	pile
$(c_0,e_0).e$	Acc 0;c	*s*	e_0	c_0	s
$(c_0,e_0).e$	Acc n+1; c	*s*	*e*	Acc n;c	*s*
e	Push(c'),c	s	e	c	(c',e).s
e	Grab;c	$(c_0,e_0).s$	$(c_0,e_0).e$	c	*s*
e	Grab;c	$<c_0,e_0>.S$	e_0	c_0	*(Grab;c,e).s*
e	Reduire(c');c	*s*	*e*	c'	*<c,e>.s*

Cette machine a servi de base par la suite à la machine abstraite Zinc.

2.2.5. Machine ZINC

La machine abstraite Zinc[Le96][GL02] a servi de base pour l'implantation du langage de programmation fonctionnelle Ocaml, développée par Xavier Leroy. Elle peut être vue comme une machine de krivine avec une marque en fin de pile qui permettra le traitement de l'appel par nom. Elle suit le modèle « push-enter », par opposition au modèle « eval-apply » de la SECD[Lan63] ou de la CAM[CCM87], c'est à dire que les arguments d'une fonction sont évalués et empilés de droite à gauche, puis la fonction est évaluée et appelée, son code teste

alors le nombre d'arguments fournis, continue en séquence s'il y en a assez, ou renvoie une cloture sinon. Ce modèle implémente efficacement le passage d'argument curryfié, sans aucune allocation si la fonction est complètement appliquée. Elle est construite dans l'objectif de permettre la compilation de langage ml d'une façon optimale, c'est à dire de construire moins de clotures possibles.

a- Présentation de la machine

Elle est composée comme krivine d'un pointeur ver le code, d'un registre pour contenir l'environnement courent. L'environnement est une liste de valeur, qui sont soit des clotures, représentant les fonctions, ou bien des constantes (entier, booléens, réel,..). Un accumulateur est utilisé pour contenir les résultats intermédiaires. (il n'était pas nécessaire dans la machine de krivine puisqu'il n'y avais que les clotures. Le pointeur de code ainsi que le registre d'environnement étaient suffisant pour représenter le résultat. Deux piles sont nécessaires, la première pile dite *pile argument*, contient les arguments pour l'appel de fonctions, qui sont des séquences de valeurs séparées par des marques (une valeur spéciale notée Ə). La seconde pile, *pile retour*, contient les clotures non allouées, qui sont un pointeur ver le code plus environnement.

b. Schéma de compilation

La représentation $\boldsymbol{T}$[E], E est une expression dont la valeur est la valeur du corps de la fonction en cour d'évaluation.

Accès aux variables locales

Le schéma de compilation d'une variable locale d'indexe n est :

$$\boldsymbol{T}[n]= \boldsymbol{c}[n]= \text{Access}(n)$$

Accès aux variables locales :

Code	Acc	Env	Pile. Arg	Pile retour	code	Acc	Env	Pile Arg	Pile Retour
Access(n) ;c	a	$e=v_0..v_n..$	s	r	c	v_n	e	s	r

Application

$\boldsymbol{T}[(M\ N_1..N_k)]= \boldsymbol{c}[N_k]$; Push;...; $\boldsymbol{c}[N_1]$;push; $\boldsymbol{c}$[M];Appterm

$\boldsymbol{c}[(M\ N_1..N_k)]$=Pushmark; $\boldsymbol{c}[N_k]$; Push;...; $\boldsymbol{c}[N_1]$;push; $\boldsymbol{c}$[M];Apply

Une marque (∂) est empilée dans la pile argument afin de séparer les nouveaux arguments et forcer la réduction à la forme normale faible de tête.

Code	Acc	Env	Pile. Arg	Pile retour	code	Acc	Env	Pile Arg	Pile Retour
Appterm;c_0	a=(c_1,e_1)	e_0	v.s	r	c_1	a	v.e_1	s	r
Apply;c_0	a=(c_1,e_1)	e_0	v.s	r	c_1	a	v.e_1	s	(c_0,e_1).r
Push; c_0	a	e	s	r	c_0	a	e	a.s	r
Pushmark; c_0	a	e	s	r	c_0	a	e	∂.s	r

L'abstraction

$\boldsymbol{T}$[λE]= grab ; $\boldsymbol{T}$[E]

$\boldsymbol{C}$[λE]= cur($\boldsymbol{T}$[E];Return)

L'instruction return termine l'évaluation d'un corps de fonctions. Elle se charge de vérifier :

- si la pile argument est vide (une marque est au sommet de la pile arg) alors elle détruit la marque est provoque un appel à l'appelant.
- si la pile n'est pas vide alors elle applique le résultat de la fonction aux arguments restants. Ce cas correspond au cas de l'évaluation partiel lorsque la fonction est appliquée à plus d'arguments qu'elle n'on a besoin.

Code	Acc	Env	Pile. Arg	Pile retour	code	Acc	Env	Pile Arg	Pile Retour
Cur (c_1);c_0	a	e	s	r	e_0	(c_1,e)	e	s	r
Grab;c_0	a	e_0	∂.s	(c_1,e_1).r	c_1	(c_0,e_0)	e_1	s	r
Grab;c	a	e	v.s	r	c	a	v.c	s	r
Return ;c_0	a	e_0	∂.s	(c_1,e_1).r	c_1	a	e_1	s	r
Return ;c_0	a=(c_1,e_1)	e_0	v.s	r	c_1	a	v. e_1	s	r

Stratégie d'évaluation

l'évaluation se fait de droite à gauche. Ainsi le nombre de clotures construit pour une fonction à n argument est de (n-1) contrairement à la machine de krivine qui est de n.

2.2.2. Approche par réduction de graphe et combinateurs

La réduction de graphe est une technique d'implémentation d'un système de réécriture où les termes sont représentés par le graphe de leur syntaxe abstraite et les occurrences d'une même variable sont représentées par un même nœud. Les substitutions s'effectuent en ajoutant un arc entre la variable à substituer et sa valeur de substitution. La réécriture d'un redex s'effectue en effaçant les arcs issus du nœud qui représente le redex, en construisant le graphe de la forme réduite et en ajoutant un arc entre le premier et le second.
La réduction de graphe permet le partage de représentation et peut être beaucoup améliorée en ajoutant au système de réécriture des règles redondantes qui tiennent compte des particularités des redex.
Cette technique a été exploité par certain auteurs pour proposer des schémas de réduction de lambda calcul plus efficaces. Wadsworth [wad71] réalisa un interpréteur du lambda calcul qui se base sur l'ordre normal de réduction de graphe.
Dans la réduction de graphe, l'expression est représentée par un graphe orienté (arbre). Par la suite cette technique a été utilisée pour définir des machines abstraites assez efficaces. On présentera par la suite l'une des représentatives de cette catégorie La G- machine.

La faiblesse des techniques conventionnelles d'implantation des langages fonctionnels se situe au niveau du traitement des variables libres. En effet plusieurs travaux ont été présentés afin de contourner ce problème.

2.2.3. La représentation de De Bruijin

De Bruijin par exemple a conçu un système basé sur le niveau d'abstraction des variables liées [DB72]. Le niveau d'abstraction d'une occurrence de variable liée est le nombre de nœuds lambda qui la sépare du nombre de nœud qui la déclare, lorsqu'on remonte dans l'arbre du terme.
Ainsi la notation des termes du λ-calcul est obtenue dans ce système en remplaçant toutes les variables liées par leur niveau d'abstraction. Les variables libres sont aussi représentées par des entiers pour respecter la cohérence de la notation. Des entiers supérieurs au niveau

maximal d'abstraction sont utilisés. A chaque nom de variable libre est associé un entier, auquel est ajouté le nombre d'abstractions qui séparent l'occurrence considérée de la racine. Ainsi le terme (λ x.x(λy.x y)) sera codé en (λ.0(λ.1 0)).
La représentation des expressions devient difficilement expressive.

D'autres travaux ont accès sur les représentations des variables à même niveau. C'est ce qu'on désigne par le *lambda lifting*[Joh85]. L'idée est de transformer toute les déclarations locales en déclarations globales, les variables libres sont éliminées par l'application de la β-réduction inverse. Les programmes obtenus ne contiennent que des combinateurs.
L'expression (λx.M) => ((λf.(λx.M))y
β-substitution inverse, pour chaque variable libre y dans M.
Exemple
(λ y.f(λx.y))5 => (λ y.f[λy.(λx.y)y])5 .
Cette technique de *lambda lifting* a été exploitée dans la définition de la G-machine.

2.2.4 Approche par combinateurs

Une autre approche fut suivi par Turner[Tur79] celle qu'il nomma approche par les combinateurs. Dans cette approche, les variables sont éliminés par un processus d'abstraction, l'expression initiale est transformée en une équivalente qui ne contient que les combinateurs S, K, I, B et C de la logique combinatoire[Sch24][CF58]. Ces combinateurs ont les règles de réduction suivantes :

S f g x $\rightarrow$ (f x)(g x)

K x y $\rightarrow$x

I x $\rightarrow$ x

B f g x $\rightarrow$ f (g x)

C f g x $\rightarrow$ (f x) g

L'abstractions d'une variable x dans une expression E, désigné par [x]E, se fait selon ces règles:

(λx.M) est représenté par[x] M

[x]x ≡ I

[x]y ≡ K y, si x $\notin$y.

[x](M N) ≡ S([x]M)([x]N)

Les expressions produites par cet algorithme sont par la suite optimisées par l'introduction des combinateurs B et C. Le code généré est exécuté par une machine nommée machine de réduction SKI qui en fait réalise la réduction de graphe pour des expressions en combinateurs. Cette machine est présentée comme un interpréteur récursif et utilise une pile pour mémoriser les nœuds parcourus.

La SK machine selon Kennaway[JR84] n'est pas efficace car le code produit est quatre fois plus long que le code du lambda calcul équivalent. Ce qui alourdie le temps d'exécution.

2.2.4.1 La G-machine

La G-machine[PJ90] met en œuvre l'appel par nécessité à l'aide d'un schéma push-enter. Elle transforme chaque fonction source en un (super) combinateur qui peut être vu comme une règle de réécriture de graphe. La définition de la G-machine est similaire à celle de la SECD. Elle a un nombre fini d'états et est composée de :

1 - Une pile S,
2 Un graphe G,
3 La séquence de code restant à exécuter C,
4 Une pile de couple (S,C) représentant la sauvegarde avec S la pile et C une séquence de code.

Un état de la machine est représenté par le quadruplet <S,G,C,D>. Le fonctionnement de la machine est décri par des transitions. Avant de donner l'ensemble des états de transitions expliquons la signification des annotations qui seront utilisées.

Une pile dont le sommet est n est notée n : S où S est le reste de la pile. Une pile vide est notée [].

Une séquence de code dont la première instruction est ***I*** est noté ***I :C*** où ***C*** est le reste du code. Une séquence de code vide est notée [].

Une sauvegarde dont le sommet est la paire ***(S,C)*** est notée ***(S,C) :D*** où ***D*** est le reste de la sauvegarde. La sauvegarde vide est notée [].

Les différents nœuds du graphe sont notés :

Int i un entier,

Cons n_1 n_2 un nœud Cons,

Ap n_1 n_2 un nœud d'application,

Fun k C une fonction à k arguments et dont le code est C.

Hole un nœud qui sera rempli par la suite ; de tel nœuds sont utilisés pour la construction de graphe cyclique.

La notation G[n= Ap n_1 n_2] représente un graphe dans le quel le nœud n est une application de n_1 à n_2.

2.2.4.2. Les transitions de la G-machine

- La transition appropriée pour Eval est choisie selon le type de nœud présent au sommet de la pile (le nœud n) :
  1- La première équation décrit le comportement d'Eval lorsqu'une application est au sommet de la pile. La pile et le code courants sont stockés dans la sauvegarde, une nouvelle pile dont le seul élément est le sommet de l'ancienne pile est formée, et Unwind est exécutée.
  2- La seconde équation donne le comportement d'Eval lorsque le sommet de la pile contient une fonction d'arité nulle. Dans ce cas, la machine sauvegarde son état, forme une nouvelle pile dont le seul élément est la Fac puis exécute le code qui lui est associé.
  3- La troisième équation donne le comportement d'Eval lorsque le sommet de la pile contient un entier. Dans ce cas, la machine ne fait rien, Ce même comportement s'applique aussi bien à Cons qu'à un nœud de fonction qui n'est pas une Fac.
- Les transitions pour Unwind se distinguent selon les quatre cas possibles :
  1- Le sommet de la pile contient un entier ou un cons. Dans ce cas, il doit être le seul élément de la pile et l'expression en cours d'évaluation est FFNT. Unwind termine donc l'évaluation en rétablissant la pile et le code sauvegardés et en mettant le résultat de l'évaluation au sommet de la pile.
  2- L'élément au sommet de la pile est (un pointeur vers) un nœud d'application. Dans ce cas, la tête de l'application est mise sur la pile et la répétition de l'instruction Unwind.
  3- Le sommet de la pile est une fonction et un nombre suffisant d'argument sont sur la pile. Dans ce cas, la pile et réarrangée puis l'exécution de la fonction va avoir lieu.
  4- Le sommet de la pile est une fonction, mais pas assez d'arguments sont sur la pile(c'est la signification de a<k). L'expression en cours d'exécution est donc en FFNT, et Uwind termine l'évaluation en rétablissant la pile et le code sauvegardés, et en mettant le résultat de l'évaluation au sommet de la pile.

La figure ci-dessous résume les instructions du G-code

Eval <v :s,G[v=APP v' n], Eval :C,D> =><v :[],G, Unwind:[],(S,C):D >

<n :s,G[n=Fun 0 C'], Eval :C,D> =><n:[],G, C:[],(S,C):D >

<n :s,G[n=INT i], Eval :C,D> =><n:s,G, C,D >

et similairement pour les nœuds CONS et les nœuds FUN qui ne sont pas des FAC

Unwind <n :[],G[n=INT i], Unwind :[],(S,C),D> =><n:S,G,C,D > et similairement pour les nœuds CONS .

<v :s,G[v=APP v' n], Unwind :[],D> =><v' :v:S,G, Unwind:[],D >

<v_0: v_1 :…: v_k:S], G[v_0=FUN k C, v_i =APP v_{i-1} n_i, (1≤i≤k),unwind:[], D >

⇨ < n_1 : n_2 …: n_k: v_k:S,G,C,D>

<v_0: v_1 :…: v_a:[], G[v_0=FUN k C',unwind:[],(S,C):D >

{A<K}=> <v_a:S,G,C,D>

Return <v_0: v_1 :…: v_k:[], G,Return:[],(S,C):D > =>< v_k:S,G,C,D>

Jump <S,G,Jump L:…:Label L: C,D > =><S,G,C,D>

Fig : Transition de la G-machine

Push < n_0 : n_1 …: n_k: S,G,Push k:C,D> => < n_k:n_0 : n_1:…: n_k: S,G,C,D>

Pushint< S,G,Pushint i:C,D> => < n: S,G[n=Int i],C,D>

Pushglobal de façon similaire.

Pop < n_1 : n_2 …: n_k: S,G,Pop k:C,D> => < S,G,C,D>

Slide < n_0 : n_1 …: n_k: S,G,Slide k:C,D> => < n_0: S,G,C,D>

Update < n_0 : n_1 …: n_k: S,G,Update k:C,D> => < n_1 : n_2:..: n_k: S,G[n_k=n_0],C,D>

Alloc < S,G,Alloc k:C,D> => < n_1 : n_2:..: n_k: S,G[n_1=Hole,...,n_k],C,D>

Head < n:S,G[n=Cons n_1 n_2],Head D:C,D> => <n_1:S,G,C,D>

Neg < n:S,G[n=Int i],Neg:C,D> => <n':S,G[n'=Int (-i)],C,D>

Add < n_1:n_2:S,G[n_1=Int n_1, n_2=Int n_2],Add:C,D> => <n:S,G[n=Int n_1+n_2)],C,D>

Mkap < n:m:S,G,Mkap:C,D> => <n:S,G[n=Ap n m],C,D>

Cons de la même façon

3-Modèles d'exécution pour la β-réduction forte

Le λ-calcul, et la β-réduction, ne servent pas uniquement de base pour l'implémentation de langages fonctionnels. Ils sont aussi utilisés dans le domaine de la démonstration automatique. En fait dans les outils d'aide à la démonstration, il est souvent nécessaire de comparer des λ-termes entre eux. Pour cela il est important de pouvoir les normaliser fortement. A ce sujet, des modèles d'exécution de la β-réduction forte sont mis en œuvre. Dans ce qui suit, on va présenter des machines définies pour cet effet.

3.1 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 λυ-calcul[LR93]. À chaque règle du λυ-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 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 (Fig 3.1). 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 (↑) appliqués.

état = terme x environnement x pile

environnement = liste de (fermetures u ↑) x N

pile = liste de fermetures

fermeture = terme x environnement

Fig 3.1 Structures de la U-machine

$(a\ b,e,p) \rightarrow_U (a, e, <b, e>:: p)$	(APP)
$(\lambda a, e, <b, e'>:: p) \rightarrow_U (a, Lift\_env\ (e) \oplus [(<b, e'>, 0)], p)$	(LBA-BET)
$(1, (c, i + 1)::e, p) \rightarrow_U (1, e, p)$	(FVARLIFT)
$(n + 1), (c, i + 1)::e, p) \rightarrow_U (n, (c, i) ::(\uparrow, 0) :: e, p)$	(RVARLIFT_)
$(1, (<a, e>, 0) :: e', p) \rightarrow_U (a, e \oplus e', p)$	(FVAR)
$(n + 1, (<a, e>, 0) :: e', p) \rightarrow_U (n, e', p)$	(RVAR)
$(n, (\uparrow, 0) :: e, p) \rightarrow_U (n + 1, e, p)$	(VARSHIFT)

Fig 3.2: Les états de la U-machine

La U-machine est décrite par le système de règles de la figure (Fig 3.2), où Lift_env, équivalent du lift du λv-calcul (↑), est défini par

$$Lift\_env\ ([\ ]) \rightarrow [\ ]$$

$$\text{Lift\_env}((c, i) :: e) \rightarrow (c, i + 1) :: \text{Lift\_env}(e)$$

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

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

$$\frac{(a, e, [\,]) \rightarrow_U^* (\lambda b, e', [\,])\ \ (b, \text{Lift\_env}(e')) \rightarrow_{nf} c}{(a, e) \rightarrow_{nf} \lambda c} \quad (L)$$

$$\frac{(a, e, [\,]) \rightarrow_U^* (n, [\,], [\langle b_1, e_1\rangle;\ldots;\langle b_q, e_q\rangle])\ \langle b_i, e_i\rangle \rightarrow_{nf} c_i}{(a, e) \rightarrow_{nf} n\ c_1 \ldots c_q} \quad (V)$$

Fig. 3.3 - 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.

La règle (V) permet de calculer les formes normales des arguments qui sont dans la pile. Ces arguments n'appartenant à aucun radical, lorsqu'on a atteint la condition d'arrêt, variable dans un environnement vide.

3.1.2 Conclusion

La U-machine permet d'obtenir la forme normale forte de λ-termes mais le temps d'exécution est assez important. L'optimisation de cette machine a été l'objectif d'une autre implémentation, nommé la UP machine.

3.2 la UP-machine

La UP-machine est définie dans[FL94]. C'est une machine équivalente à la U-machine, mais dont la stratégie d'évaluation des termes est paresseuse. Ceci par l'implantation du partage des arguments des fonctions (fermetures). A cet effet, des marques sont introduites pour isoler l'évaluation des fermetures lors de leur utilisation en tant qu'arguments, pour les réutiliser ultérieurement, dans un autre contexte.

3.2.1 Définition de UP

La UP machine est définie à partir de la U-machine. Avec les modifications suivantes :

- Un champ état mémoire est ajouté sur les états, contenant des adresses auxquelles sont associées des fermetures ;
- Les fermetures sont remplacées par leur référence dans l'état mémoire ;
- des marques sont introduites pour isoler l'évaluation des arguments de leur contexte. L'adresse à mettre à jour à la fin de l'évaluation est mémorisée.

$(bc, e, p, s) \rightarrow_{UP} (b, e, \#a :: p, s[a \leftarrow <c, e>])$	(APP)
où a = nouveau(s)	
$(\lambda c, e, \#a :: p, s) \rightarrow_{UP} (c, Lift\_env\ (e) \oplus [(\#a, 0)], p, s)$	(LBA-BET)
$(1, (c, i + 1)::e, p, s) \rightarrow_{UP} (1, e, p, s)$	(FVARLIFT)
$(n + 1, (c, i + 1)::e, p, s) \rightarrow_{UP} (n, (c, i) ::(\uparrow, 0) :: e, p, s)$	(RVARLIFT_)
$(1, (\#a, 0) :: e', p, s) \rightarrow_{UP} (c, e, M(a, e') ::p, s)$	(FVAR)
où $<c, e>= s(a)$	
$(n + 1, (\#a, 0) :: e, p, s) \rightarrow_{UP} (n, e, p, s)$	(RVAR)
$(n, (\uparrow, 0) :: e, p, s) \rightarrow_{UP} (n + 1, e, p, s)$	(VARSHIFT)
$(\lambda c, e, M(a, e') ::p, s) \rightarrow_{UP} (\lambda c, e \oplus e', p, s[a \leftarrow <\lambda c, e>])$	(LMK)

Fig. 3.4 : les règles de la UP-machine

Les structures de la UP machine sont données dans la figure 5.4 et les règles dans la figure 5.5 où Lift_env est défini par

$Lift\_env\ ([\,]) \rightarrow [\,]$

$Lift\_env\ ((c, i) ::e) \rightarrow (c, i + 1):: Lift\_env\ (e)$

et l'opérateur ⊕ 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,

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

- la fonction nouveau associe une adresse à un état mémoire : soit s un état mémoire, nouveau(s) n'appartient pas au domaine de s.

- La règle (APP), introduit les nouvelles adresses, une association [adresse fermeture] est crée 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, exception faite des règles (FVAR) et (LMK).
- 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.
- La règle (LMK) effectue la mise à jour de l'état mémoire et la restauration de l'environnement.

3.2.2 Calcul de la forme normale

L'opérateur nf qui permet le calcul de la forme normale forte des termes est définie par :

Les conditions d'arrêt sont les mêmes que la U machine :

- on se trouve dans un état de la forme (λa; e, [ ], s) ;
- on se trouve dans un état de la forme (n, [ ], p, s).

Des règles qui ont le même effet que les règles de la U-machine sont réalisées. Pour le calcul de la forme normale des termes sous les λ ne faisant partie d'aucun radical.

La règle (L) se retrouve dans la UP (voir figure 3.5). Par contre, pour le calcul de la forme normale des termes présents dans la pile quand on est dans un état de la forme (n,[ ], p, s), certaines transformations sont réalisées :

Dans la pile, il y a des marques en plus des adresses de fermetures. 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. Il faut procéder à un dépilement progressif. Pour cela un opérateur $\rightarrow_{dep}$ est introduit et permet de

retrouver la forme normale du terme représenté par un état de la forme (n, [ ], p, s') (voir figure 3.5).

dep a été définie dans le but de transformer le couple (terme, pile) en un terme correct en forme normale. La pile contenant des adresses et des marques.

Pour la définition de dep, deux règles sont définies: la première pour dépiler une adresse, la deuxième pour dépiler une marque. On obtient alors le système de la figure 3.6.

- 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.

Note : 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 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 (n) et les suivants étant en forme normale.

$$\frac{(b, e, p, s) \rightarrow^{*}_{UP} (\lambda c, e', [\,], s') \quad (c, Lift\_env\,(e'), [\,], s') \rightarrow_{nf} d}{(b, e, p, s) \rightarrow_{nf} \lambda d} \quad (L)$$

$$\frac{(b, e, p, s) \rightarrow^{*}_{UP} (n, [\,], p, s') \quad (n, p, s') \rightarrow^{*}_{dep} (d, [\,], s'')}{(b, e, p, s) \rightarrow_{nf} d} \quad (V)$$

Fig. 3.5 - Règles de la UP-machine pour le calcul de la forme normale

$$\frac{(c, e, [\,], s) \rightarrow_{nf} d \text{ où } \langle c, e\rangle = s(a)}{(b, \#a :: p, s) \rightarrow_{dep} (bd, p, s[a \leftarrow \langle d, [\,]\rangle])} \quad (DEPADR)$$

$$\frac{(b, e, p, s[a \leftarrow \langle b, [\,]\rangle]) \rightarrow_{nf} d}{(b, M(a, e) :: p, s) \rightarrow_{dep} (d, [\,], s[a \leftarrow \langle b, [\,]\rangle])} \quad (DEPMK)$$

Fig. 3.6 - définition de l'opérateur dep

3.2.3 Conclusion :

La U-machine et la UP-machine réalisent toute les deux de l'évaluation forte. Le temps de calcul de la UP-machine est meilleure que celui de la U-machine mais selon l'auteur ça reste long. L'auteur propose de se tourner vers des implémentation par réduction de graphe.

3.3 un évaluateur de la β–réduction forte

Un évaluateur de la β–réduction forte à été défini par X.Leroy et B.grégoire dans [LG02]. Cette implémentation a été réalisée, afin d'améliorer les performances du prouveur coq.
Cette implémentation utilise une machine réalisée au départ pour l'implantation du langage fonctionnel (Ocaml), la ZINC machine[Le96].
On a vue que la ZINC machine s'arrête à la faible forme normale de tête. L'idée de cette implémentation est d'utiliser cette machine (la Zinc) comme un module pour l'évaluation faible. Une fois la forme réduite est obtenu, une seconde réduction est réalisée par un second module. Le second module s'occupe donc de la réduction de termes en faible forme normale de tête en forme normale forte. Les termes considérés sont tous des termes dont la forme normale existe c'est à dire la réduction termine. Cette clause est assuré par le système de type du prouveur COQ.
Les transition de la machine abstraite sont données à la figure (Fig 3.7)

Code	Env	Pile	#args
ACC(i); k k	e e	s e(i).s	n n
CLOSURE(c); k k	e e	s [c.e].s	n n
PUSHRETADDR(c); k k	e e	s < c, e,n>.s	n n
APPLY(i) C'	e e'	[c:e]:s s	n i
GRAB(i); k k	e $v_1 \ldots v_i$:e	v i... v 1 .s s	n ≥ i n - i
GRAB(i); k C'	e e'	vn…v 1 . < c' , e',n'> .s [(GRAB(i - n); k).v 1...vn.e].s	n < i n'
RETURN C'	e e'	v.<c',e',n'> .s v.s	0 n'
RETURN C'	e e'	[c'.e']:s s	n > 0 n
MAKEBLOK(t,m);C C	e e	v1…v m.s [T: v1…v m].s	n n

SWITCH ($C_{1,...,}C_M$) C_T	e $v_1 \ldots v_P$	[T: v1…v p].s s	n 0
C_0= SWITCH ($C_{1,...,}C_M$) RETURN	e e	[0:ACC,k].s [0:ACC,[2:k,c_0,e]].s	n 0
CLOSUREREC(c');c c	e e	s v.s	n n où v=[T_λ :c',v.e]
GRABREC;c	e	[0:ACC,k].s [0:ACC,[3:c,e,k]].s	n+1 n
c_0= GRABREC;c c'	e e'	<c',e',n'> [T_λ :c_0,e] ;.s	0 n'

fig 3.7 Les états de transitions de la machine

4-Le λ–calcul à substitution explicité et les systèmes de réécritures de termes

Les λ-calculs avec substitutions explicites ont été introduits récemment. Ils peuvent être classés en deux catégories : Le λ–calcul avec substitution explicite utilisant la notation de **De bruijin**[DB72] en cite entre autre : le λσ-calcul proposé par Abadi, Cardelli, Curien et Lévy [ACCL90], le λσ⇓-calcul proposé par Hardin et Lévy [HL89, CHL92], le τ-calcul [Río93], et enfin le λν-calcul (lire lambda upsilon) proposé par Pierre Lescanne [Les94, LR93]. La deuxième catégorie concerne les λ–calcul avec substitution explicite utilisant les noms : Le système révers, le système de rose et puis le lambda Cβ+ calcul. 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.

4.1 Les λ-calcul avec substitution explicite avec nom

On a vu dans le chapitreI que l'opération de substitution est coûteuse en temps. La recherche des occurrences à substituer ainsi que l'identification elle-même de ces variables est une opération complexe. Afin de remédier à ces problèmes, plusieurs auteurs ont proposé des formalismes qui se basent essentiellement sur l'axiomatisation de la β-réduction permettant d'éviter les actions de renommage. On trouve les λ-calcul avec substitution explicite avec nom. La particularité de ces calculs est qu'ils utilisent des noms pour désigner les variables. La plus part des formalisations des théories de types ou lambda calcul utilisent la notation de **De Bruijin** pour éviter de passer par l'alpha conversion, afin d'éviter la capture de variables. On donne ci-dessous un exemple de chacun des systèmes : à substitution explicite avec indice ; et puis à substitution explicite avec nom.

4.2 Les λ-calcul avec substitution explicite avec indice

4.2.1 Le λν-calcul

Le λν-calcul[Les94, LR93] est un calcul avec substitutions explicites qui contient un ensemble minimal d'opérateurs.

Il y a quatre opérateurs sur les termes, abstraction (λ-), application (- -),fermeture -[-] et variables (n), et trois opérateurs sur les substitutions, slash (-/),lift (⇑ (-)) et shift (↑). L'opérateur fermeture introduit les substitutions dans le calcul. λν utilise la notation de De Bruijn pour les variables qui sont notées 1, 2,…, n, n + 1…. Un terme qui ne contient pas de fermetures est appelé un terme pur.

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

Termes	a	::= n | a b | λa | a [s]
Substitutions	s	::= a /|⇑ (s) | ↑
Variables	n	::= 1 | n+ 1

Fig. 3.1 - Grammaire des termes du λν-calcul

(Beta)	(λa) b	→	a [b/]
(App)	(a b) [s]	→	a [s] b [s]
(Lambda)	(λa) [s]	→	λ(a [⇑ (s)])
(Fvar)	1 [a/]	→	a
(Rvar)	n+ 1[a/]	→	n
(Fvarlift)	1 [⇑ (s)]	→	1
(Rvarlift)	n + 1 [⇑ (s)]	→	n [s][↑]
(Varshift)	n [↑]	→	n+ 1

Fig. 3.2 - Le système λν

La règle (Beta) est la règle de β-réduction classique (voir chapitre 1 section 2.6). 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 (1, puis 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 ⇑, et indirectement de l'opérateur ↑, introduit par ⇑ (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 (⇑) et shift (↑).

Intuitivement b/; ⇑ (s) et ↑ ont la signification suivante :

b/ :
$$\underline{1} \rightarrow b$$
$$\underline{2} \rightarrow \underline{1}$$
$$\vdots$$
$$\underline{n+1} \rightarrow \underline{n}$$

⇑ (s) :
$$\underline{1} \rightarrow \underline{1}$$
$$\underline{2} \rightarrow s(\underline{1})\ [\uparrow]$$
$$\vdots$$
$$\underline{n} \rightarrow s(\underline{n})\ [\uparrow]$$

↑ :
$$\underline{1} \rightarrow \underline{2}$$
$$\underline{2} \rightarrow \underline{3}$$
$$\vdots$$
$$\underline{n} \rightarrow \underline{n+1}...$$
$$\vdots$$

Ainsi, le mécanisme de substitution est décrit au même niveau que le mécanisme de réduction. Ce système a servi de base pour l'implémentation de la U-machine(chapitre II, section 3.1) et la UP-machine (chapitre II, section 3.2).

Ce calcul se base donc sur la notation de **De Bruijin,** bien que cette notation soit plus pratique au niveau de l'implémentation, il est beaucoup plus naturel de travailler directement avec des noms de variables.

4.2.2 Le système λxgc

Le λx a été proposé par Kristoffer Rose dans [KRI96]. C'est un calcul avec nom.

Les termes λx sont formés à partir de la grammaire suivante:

M,N ::= λx.M | MN | x |

M ⟨ x := N⟩

Le terme M[x] ⟨ x := N⟩ correspond au terme dans le quel la substitution de x par N est explicite, M[x] ⟨ x := N⟩ représente le terme dans lequel, dans le futur, x sera remplacé par le terme N dans M. Il est défini par les règles suivantes :

$(\lambda x.M[x])N \to_{\lambda x} M[x]\langle x := N\rangle$ (b)

$(M_1[x]M_2[x])\langle x := N\rangle \to_{\lambda x} M_1[x]\langle x := N\rangle\, M_2[x]\langle x := N\rangle$ (xap)

$(\lambda y.M[x,y])\langle x := N\rangle \to_{\lambda x} \lambda y.M[x,y]\langle x := N\rangle$ (xab)

$x\langle x := N\rangle \to_{\lambda x} N$ (xv)

$M\langle x := N\rangle \to_{\lambda x} M$ (xgc)

La règle b initialise la β-réduction.

Les pas de β-réductions peuvent être entremêlés, contrairement au lambda calcul où substitution est réalisée jusqu'au bout après chaque pas d'élimination de β-redex. En plus dans ce système il y a introduction de nouveau symboles inconnus au système de λ-calcul.

4.2.3 Le système λCβ^+ -calcul

4.2.3.1 Introduction

Le système λCβ^+ calcul[GMMS01] est un système de réécriture de terme qui permet de réaliser la β-réduction sans avoir recoure à aucune α-conversion, en effet la β-réduction est réalisée par un certain nombre de règles de réécriture qui permettent d'aboutir à la forme normale s'elle existe. Le système λCβ^+ calcul enrichi le système λcalcul classique par l'introduction de combinateurs, La β-réduction est éclatée, les variables libres sont renommées au cours du processus de substitution par l'introduction de combinateurs pour abstraire les variables libres.

4.2.3.2 λ*C*β^+ *-terme*

Les termes de λCβ^+ calcul sont définies inductivement par :

1) Les variables x,y,z,…, sont des λCβ^+ termes,
2) Les Constantes S, K et I sont des λCβ^+ termes,
3) Si M est un λCβ^+ terme alors λx.M et λx*.M sont des λCβ^+ termes,
4) Si M et N sont des λCβ^+ termes alors (M N) est un λCβ^+ terme.

NB : Les termes générés par le λCβ^+ calcul englobent les termes du λ-calcul classique. Les termes du λCβ^+ comprennent donc l'ensemble du λ-terme ainsi qu'un ensemble d'autre terme de la forme λx*.M avec M un terme de λCβ^+, λx.S x…

Les règles de réécritures sont données dans la figure 1.

4.2.3.3 La réduction dans λ*C*β+ *-calcul*

Les règles (λC, λβ_1, λβ_2,λβ_3, λβ_4) sont les règles de bases qui réalisent la réduction.

- La première règle (λC) simule l'opération de renommage(des variables liées du sous terme M) qui risque de causer la capture des variables libres (dont le nom est y) de N s'elles figurent. par l'application de l'algorithme λ*abstrait.
- Les règles (λβ_1)-(λβ_4) éclatent systématiquement les termes complexes (application) en termes plus simples sur qui peuvent être appliqués les règles (β_1)- (β_2).

L'algorithme λ*abstrait est représenté par les règles (β_1*)- (β_4^*). Cet algorithme permet d'abstraire les variables liées qui risque de causer la capture des variables libres. Son application introduit des termes avec combinateurs.

Les règles de réécritures du système $\lambda C\beta^+$ calcul, évite les opérations de renommage par l'utilisation de combinateurs et l'introduction de nouvelles variables g_i, $I \in \{0,1,2,\ldots,n,\ldots\}$.

Exemple

Dans l'exemple précédent (§ 1.3) le terme $(\lambda x.(\lambda y.x))y$ est réduit en sa forme correcte par l'utilisation de l'α-conversion, par contre dans le système $\lambda C\beta^+$ calcul, la réduction se fait comme suit :

$(\lambda x\ (\lambda y.x))y \rightarrow\ (\lambda x\ (\lambda^* y.x))y$ (* règle λC *)

$(\lambda x\ (\lambda^* y.x))y \rightarrow (\lambda x\ (Kx))y$ (* règle β_2* *)

$(\lambda x\ (Kx))y \rightarrow (\lambda g\ .(\lambda x\ .x)y)$ (* règle $\lambda\beta_2$ *)

$(\lambda g\ .(\lambda x\ .x)y) \rightarrow (\lambda g\ .y)$ (* règle $\lambda\beta_1$ *)

Le renommage de la variable y dans le terme $(\lambda y.x)$ est réalisé implicitement. Les règles de réécritures doivent être appliquées dans l'ordre, la règle $(\lambda\beta_2)$ ne peut s'appliquer que si la règle $(\lambda\beta_1)$ ne peut s'appliquer de même pour les autres règles. Les règles doivent être filtrées dans l'ordre.

La β-réduction est réalisée pas à pas ce qui n'était pas possible dans le λ-calcul classique.

4.2.3.4 Les règles de réécritures de $\lambda C\beta^+$

(β_1) $((\lambda x.\ x)\ N)$ -> (N)
(β_2) $((\lambda x.\ y)\ N)$ -> (y) si $x \notin y$
(λC) $((\lambda x.\ \lambda y.\ M)\ N)$ -> $((\lambda x.\ (\lambda y^*.M))\ N)$
(β_{1^*}) $(\lambda x^*.\ x)$ -> I
(β_{2^*}) $(\lambda x^*.\ y)$ -> K y
(β_{3^*}) $((\lambda x^*.\ \lambda y.\ M)$ -> $(\lambda^* x.(\lambda y^*.M))$
(β_{2^*}) $((\lambda x^*.M\ N)$ -> S $(\lambda x.M)$ $(\lambda x.N)$
$(\lambda\beta_1)$ $((\lambda x.\ S\ P\ Q)\ N)$ -> $\lambda g.\ ((\lambda x.P)Ng\ ((\lambda x.Q)Ng)$
$(\lambda\beta_2)$ $((\lambda x.\ K\ P)\ N)$ -> $\lambda g.\ ((\lambda x.P)N)$
$(\lambda\beta_3)$ $((\lambda x.\ S\ P\ )\ N)$ -> $\lambda g.\ \lambda g_1.\ ((\lambda x.P)Ng_1\ (gg_1))$
$(\lambda\beta_4)$ $((\lambda x.\ P\ Q\ )\ N)$ -> $((\lambda x.P)N((\lambda x.Q)N)$
(λK) (K) -> $\lambda g\ g_1.g$
(λI) (I) -> $\lambda g\ .g$
(λS) (S) -> $\lambda g\ g_1g_2.gg_2(g_1g_2)$

Figure 1 : Les règles de réécritures de $\lambda C\beta^+$ calcul

4.2.3.5 Propriétés

a – La confluence :

Le système $\lambda C\beta^{+}$ calcul, préserve la forte normalisation du λ-calcul, voir pour la démonstration [GMMS01].

b- simulation pas à pas de la β-réduction :

Le système $\lambda C\beta^{+}$ calcul, permet de réaliser l'opération de réduction pas à pas.

c- Traitement des renommage :

Le problème de renommage de variable est prie en charge dans le système lui même.

L'opération se fait d'une façon implicite.

4.2.4 . Conclusion

Dans ce qui suit, on se propose de réaliser une machine abstraite implémentant ce système $\lambda C\beta^{+}$ calcul.

Chapitre III

Mise en œuvre de la Cβ+Machine

Dans ce chapitre on se propose de définir une machine abstraite qui se basera sur le λCβ+calcul. Dans le chapitre II on a présenté certaines machines abstraites. On a vu comment chaque machine traite le problème des variables libres. La première machine (SECD[LAN63]) ignore simplement ce problème et donc n'accepte que des termes clos. D'autre par contre y ont recoure soit à des théorie équivalente au λ-calcul (La logique combinatoire pour la SK machine, la logique combinatoire catégorique pour la machine CAM[]) soit à un système de codage des variables (notation de Bruijin[DB72]) pour les machines (Krivine[], Krivine avec marque[], Zinc[Le96],…). Toutes ces machines abstraites ne réalisent que de l'évaluation faible. On a vue que l'évaluation faible est le mécanisme utilisé par les langages de programmation fonctionnelle et ces machine abstraites ont été définie pour servir de base pour l'implémentation de langages fonctionnels. Nous, ce qu'on se propose, c'est de réaliser une machine abstraite qui fera de l'évaluation forte. 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.

Dans ce qui suit on présentera la mise en œuvre d'une machine abstraite qu'on nommera $\lambda C\beta^+$ machine. La $\lambda C\beta^+$ machine sera définie donc dans l'objectif de calculer la forme normale forte des expressions du λ-calcul (lorsque cette forme normale existe).

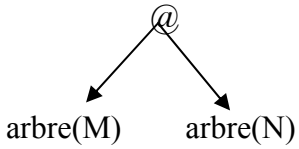
3.1 - Réalisation

On va essayer de définir notre machine abstraite. Le choix d'une machine abstraite nous permet d'être indépendant d'une machine particulière.

A présent on doit définir comment on va implémenter cette machine.

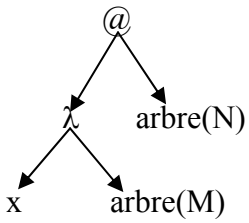
On va adopter la méthode par réduction de graphe. Les expressions seront donc représentées par un graphe.

Ainsi l'expression de la forme (M N) peut se représenter par



Le signe "@" est appelé marque du nœud et indique qu'il s'agit d'une application.

l'expression de la forme ((λx.M) N) peut se représenter par

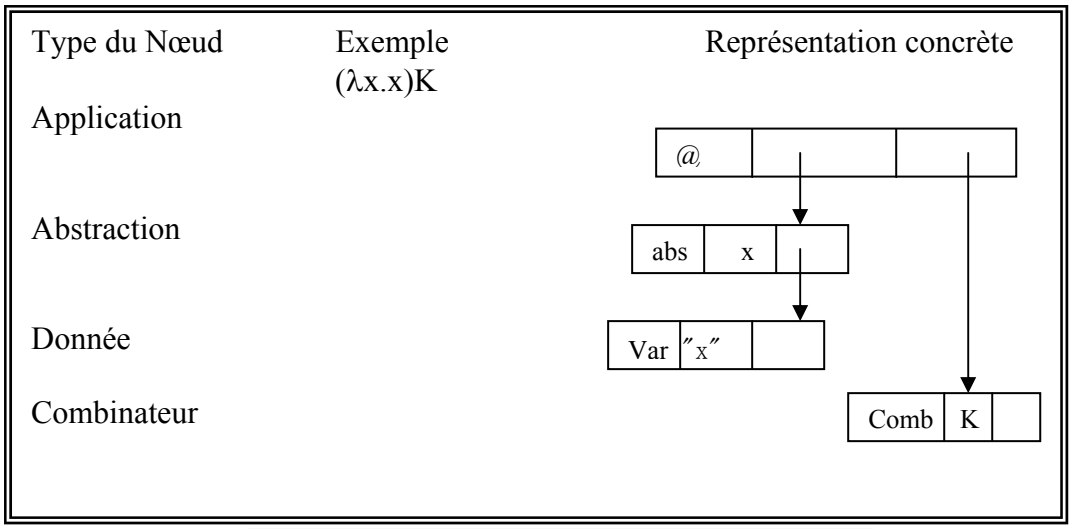


Le graphe d'une expressions est caractérisé par les nœuds et les feuilles. Les nœuds se diffèrent selon la nature des expressions. L'expression(M N) est une application, le nœud représentant cette expression est de type application symbolisé par (@), par contre l'expression (λx.x) est une abstraction, le nœud la représentant est de type abstraction (λ). Au niveau des feuilles on aura la donnée simple (variable, combinateur, ..)

Plus concrètement, chaque nœud du graphe va être représenté par trois champs :

Marque	Champ 2	Champ 3

- Champ N°1 : On l'appellera Marque, représentera le type du nœud (donnée, abstraction, combinateur, application).
- Champ N°2, N°3: Contiendra soit un pointeur vers d'autre champs soit la valeur du nœud lorsqu'il s'agirai d'une donnée simple



Représentation concrète d'une expression

Maintenant on définie notre machine. Tout comme la SECD, la Cβ+ machine est une machine à transitions d'état et possede un nombre fini d'état. Elle est composée de quatre registres.

- Une pile S1 : Mémoriser le chemin du graphe en cours de réduction ;
- Un graphe G qui représentera l'expression à réduire ;
- Une pile S2 Sous forme de liste: Mémoriser les variables générées dans le cas d'instruction de type(a,b,c) ou bien quand le terme est faiblement réduit et peut être fortement réduit ;
- Une pile S3 (pile sauvegarde) pour garder une partie du code, ou la partie gauche du terme réduit quand il s'agit d'une application. (@(p, q) ; le terme p réduit).

Avant de présenter les états de transition de la machine, on présente la notation qu'on utilisera.

3.1.2 - Notation

Une pile dont le sommet est N est notée N **:**S où S est le reste de la pile.

Une pile vide est notée []**:**S.

Une sauvegarde sera notée (S1,S2) :S3 , S1 est une partie du terme et S2 contient les variables.

Les différents types de nœuds représentant un terme sont :

DATA x x est une donnée simple

COMB comb comb est un combinateur (S,K,I)

APP p q Un nœud application ; dans la suite on utilisera le symbole @ au lieu de APP, pour alléger les écritures.

ABS x M Un nœud abstraction ;

G [n := APP p q] : Représente le nœud d'application d'un nœud p au nœud q. (Les noms p, q, représentent la référence vers les nœuds)

ABST (x,M) : La fonction qui permet d'abstraire la variable x, du terme M.

La fonction ABST est définie comme suit

```
ABST (x,x) ->α1 COMB I
ABST (x,y) ->α2 @(COMB K, y)
ABST (x,abs(y.M)) ->α3 ABST(x,ABST(y,M))
ABST ((x,@(M,N)) ->α4@(@( COMB S ,abs(x,M)), abs(x,N))
```

Fonction d'abstraction ABST

3.1.3 - Etat de transition de la Cβ+Machine

La machine est décrite par ses états de transition qui se présentent comme suite :

A_0 :<$\$_1$, @(p,q),$\$_3$,$\$_4$> ⇒< q:$\$_1$, p,$\$_3$,$\$_4$>

A_1 :<p:$\$_1$, abs(x,x),$\$_3$,$\$_4$> ⇒< $\$_1$, p,$\$_3$,$\$_4$>

A_2 :<q:$\$_1$, abs(x,p),$\$_3$,$\$_4$> ⇒<$\$_1$, p,$\$_3$,$\$_4$>(p un λ-terme avec x∉ p)

A_3 :<$\$_1$, abs(x.(abs(y,p)),$\$_3$,$\$_4$> ⇒< $\$_1$,abs(x,ABST(y,p)),$\$_3$,$\$_4$>

A_4 :<n_1:$\$_1$, abs(x,@(@(COMB S,p),q)),$\$_3$,$\$_4$>

⇒<[],@(@(@(abs(x,p),n_1),g_0),@(@(abs(x,q),n_1),g_0)),g_0,($\$_1$,$\$_3$):$4>

A_5: <n_1:$\$_1$, abs(x,@(COMB K,p)),$\$_3$,$\$_4$> ⇒

< n_1, abs(x,p),g_0,($1,$3):$4>

A_6 : <n_1:$\$_1$, abs(x,@(COMB S,p)),$\$_3$,$\$_4$> ⇒

< []:$\$_1$, @(@(@(abs(x,p), n_1),g_1), @(g_0,g_1)),g_1:g_0,($\$_1$,$\$_3$):$4>

A_7 : < n:$\$_1$, abs(x,@(p,q)),$\$_3$,$\$_4$> ⇒

<@(abs(x,q),n)):$\$_1$, (@(abs(x,p),n) ,$\$_3$,$\$_4$>

A_8 : < p:$\$_1$, DATA q,$\$_3$,$\$_4$> ⇒ < $\$_1$, p,$\$_3$,m:q:$\$_4$> (m une marque)

A_9 : < p:$\$_1$, DATA q,$\$_3$,m:$n_1$: $\$_4$> ⇒< $\$_1$,p , $\$_3$, @f($n_1$,DATA q):$\$_4$>

A_{10} : < []:$\$_1$, DATA p,$\$_3$,m:$n_1$:$\$_4$> ⇒< []:$\$_1$, @f($n_1$,p),$\$_3$,$\$_4$>

A_{11} : < []:$\$_1$, DATA q,v:$\$_3$,$\$_4$> ⇒ < []:$\$_1$, abs(v,DATA q),$\$_3$,$\$_4$>

A_{12} : < []:$\$_1$, DATA q,[]:$\$_3$,(a,v):$\$_4$> ⇒ < a:$\$_1$, DATA q,v:$\$_3$,$\$_4$>

A_{13} : < []:$\$_1$,abs(x,@(p,q)),v:$\$_3$,$\$_4$> ⇒ < q:$\$_1$,p,x:v:$3,$\$_4$>

A_{14}: < n:$\$_1$,COMB S,$3,$\$_4$> ⇒

< n:$\$_1$, abs($g_0$,(abs($g_1$,(abs($g_2$,@(@ ($g_0$,$g_2$), @ ($g_1$,$g_2$)))))))),$3,$\$_4$>

A_{15} : < n:$\$_1$,COMB K,$3,$\$_4$> ⇒ < n:$\$_1$, abs($g_0$,abs($g_1$,$g_0$),$3,$\$_4$>

A_{16} : < n:$\$_1$, COMB I,$3,$\$_4$> ⇒ < n:$\$_1$,abs($g_0$,$g_0$),$3,$\$_4$>

Règle de transitions de la Cβ+ machine

Remarque : Les variables g_0, g_1, g_2…..doivent être distinctes de toutes les variables figurant dans un terme. Pour respecter cette clause, on pourai définir une fonction qui produirai automatiquement ces variables, comme c'est fait par certains auteurs. Il est donc plus juste d'écrire par exemple dans la règle A_4 : g_0 := *call* gen_var() avec gen_var() la fonction qui produit les variables g_i.

3.1.4 Ordre de réduction

Pour la réduction d'expression, on adopte l'ordre normal de réduction (stratégie sure). On réduit toujours le terme le plus à gauche. Un terme de type application est éclaté en termes plus simples. Le premier terme (le terme le plus à gauche) va être réduit, puis mis sur la pile S3. Le second terme au quel il est appliqué, est réduit ensuit. Puis le terme se trouvant déjà sur la pile S3 va être repris à la tête du graphe, pour continuer la réduction.

3.1.5 Fonctionnement de la Cβ+ machine

Les opérations de la Cβ+ machine se déroulent par l'application des règles de transitions ci dessus.

Le terme à réduire est pointé par la racine du graphe G. Selon la structure du terme à réduire, la machine va passer à un second état. <S1, G, S2, S3> => <S1',G',S2',S3'>.

Initialement les structures (S1, S2, S3) sont vides, G pointe vers le terme à réduire.

- S'il s'agit d'un *terme application*, l'argument est juste mis sur la pile S1.
- Si G contient une *donnée simple* (le terme est irréductible) : dans ce cas, deux cas principaux sont possibles :

♦ La pile argument S1 est vide,

- mais S2 contiens des variables :

  Le contenu de la pile S2, va être vidé en créant récursivement une abstraction

  (() :S , data x , v_0 :S, D) =>(() :S , lambda v_0.data x , S, D)

- la pile de variables S2 est vide dans ce cas,

  Si la pile S3 n'est pas vide alors

  le terme courant doit être complètement réduit, la réduction continue avec le reste du terme, en rétablissant la pile S2 et le code sauvegardés.

  Sinon (S3 est vide), la réduction s'arrête est le contenu de G est renvoyé en résultat.

♦ La pile argument S1 n'est pas vide mais (G pointe vers un terme irréductible en FFNT) : dans ce cas

1- Le terme vers qui pointe G ainsi que le contenu de la pile S2 seront Sauvés dans S3 afin de continuer la réduction.
2- Le terme vers qui pointe G est marqué par une marque avant d'être sauvé). un terme marqué se trouvant sur la pile S3 implique que ce terme est le terme à gauche du terme en cours de réduction et aussi qu'il est réduit.
3- La racine du graphe G est mise à jour par le terme qui se trouvais à la tête de la pile S1.

-Si G pointe vers un terme de type abstraction, on a plusieurs cas

♦S'il s'agit d'abstraction d'une variable avec un corps d'une variable,

soit on est devant le cas de l'identité dans ce cas la réduction continue avec celle de l'argument,

si par contre il s'agit d'un corps où la variable à abstraire ne figure pas dans le corps, dans ce cas il s'agit de continuer la réduction avec la projection du corps de l'abstraction.

♦S'il s'agit d'une abstraction d'une variable avec un corps où il y a d'autre abstraction dans ce cas l'algorithme ABST est appliqué.

Les trois dernière règles (15,16,17), permettent d'avoir des résultats, uniquement sous forme de lambda termes c'est à dire sans combinateurs.

Pour mieux expliquer le fonctionnement de la Cβ+ machine, on donne quelque exemples.

Exemple 1

Soit à réduire le terme (λx .x y)(y z).

Ce terme sera représenté par `@(abs(x,@(x,y)),@(y z))`

La réduction de ce terme se fera à la suite des états de transitions de la machine qui seront comme suite :

1-([],@(abs(x, @(x,y)),@(y z)), [],[])
=>A0([@(x,y)],abs(x, @(x,y)), [],[])
2-([@(x,y)],abs(x, @(x,y)), [],[])=>A8
([@(abs(x,y),@(x,y))],@(abs(x, x),@(x,y)), [],[])
3-([@(abs(x,y),@(x,y))],@(abs(x, x),@(x,y)), [],[])
=> A0 ([@(x,y) :@(abs(x,y),@(x,y))], abs(x,x), [],[])
4-([@(x,y) :@(abs(x,y),@(x,y))], abs(x,x), [],[])
=>A1([@(abs(x,y),@(x,y))], @(x,y), [],[])
5-([@(abs(x,y),@(x,y))], @(x,y), [],[])
=>A0([y:@(abs(x,y),@(x,y))], x, [],[])
6-([y:@(abs(x,y),@(x,y))], x, [],[])
=>A9 ([@(abs(x,y),@(x,y))], y, [],[m.x])
7-([@(abs(x,y),@(x,y))], y, [],[m.x])
=> A11([], @(abs(x,y),@(x,y)) , [],[@f(x,y)])
8- ([], @(abs(x,y),@(x,y)) , [],[@f(x,y)])
=> A0([@(x,y)], abs(x,y) , [],[@f(x,y)])
9-([@(x,y)], abs(x,y) , [],[@f(x,y)])
=>A2 ([], y , [],[m.@f(x,y)])
10- ([], y , [],[m.@f(x,y)])
=>A10 ([], @f(@f(x,y),y) , [],[])

Le résultat de la reduction du terme (λx .x y)(y z) est (xy)y

Notre machine permet donc de réaliser la réduction de λ-termes. Le terme en question est réduit pas à pas. En plus de la réduction de λ-terme, on peut aussi réduire des termes sous forme de combinateurs.

Exemple 2

Soit à réduire le terme (K x y) :

Ce terme sera représenté par @(@(comb k,x),y)

La réduction de ce terme se fera comme suite :

1- ([],@(@(comb k,x),y),[],[]) => ^{A0}(y:[],(@(comb k,x), [],[])

2- (y:[],(@(comb k,x), [],[]) => ^{A0}(x:y:[],comb k, [],[])

3- (x:y:[],comb k,[],[])=> ^{A16}(x:y:[],abs(g0,(abs(g1,g_0)),[],[] >

4- (x:y:[],abs(g0,(abs(g1,g_0)),[],[]>

=> $^{A}{}_{3}$(x:y:[],abs(g0,(abst(g1,g_0)),[],[] >

abst(g1,g_0) ->$^{\alpha 2}$ @(COMB K,g_0)

5- (x:y:[],abs(g_0, @(COMB K,g_0)),[],[]> => $^{A}{}_{6}$
(x:[],abs(g_0,g_0)), g_2:[], (y,[]):[]>

6- (x:[],abs(g_0,g_0)),g_2:[], (y,[]):[]> => $^{A}{}_{1}$

([],abs(g0,x)), g_2:[], (y,[]):[]>

7- ([],abs(g0,x)),g_2:[],(y,[]):[]>=> $^{A}{}_{2}$ <[],x,g_2:[],(y,[]):[]>

8- <[],x,g_2:[],(y,[]):[]> => $^{A}{}_{12}$ <[],abs(g_2,x),[],(y,[]):[]>

9- <[],abs(g_2,x),[],(y,[]):[]>=> $^{A}{}_{13}$< y:[],abs(g_2,x),[],[]>

10- <y :[],abs(g_2,x),[],(y,[]):[]>=> $^{A}{}_{13}$< [],x,[],[]>

Le résultat de la réduction du terme (K x y) est x. C'est bien le résultat escomté. Uniquement, si on se penche sur le processus de réduction, on réalise bien que c'est assez long. En fait ce résultat est bien connu et pour optimiser la réduction de termes sous forme de combinateurs, on pourai ajouter à la définition de la Cβ+machine, les règles de réduction des combinateurs (S, K, I).

< n_1:n_2:n_3:$\$_1$,COMB S,$3,$\$_4$> ⇒

< $\$_1$,abs($n_1$,(abs($n_2$,abs($n_3$,@(($n_1n_3$),($n_2n_3$))))),$3,$\$_4$>

< n_1:n_2:$\$_1$,COMB K,$3,$\$_4$> ⇒< $\$_1$, n_1,$3,$\$_4$>

< n_1:$\$_1$,COMB I,$3,$\$_4$> ⇒< $\$_1$, n_1,$3,$\$_4$>

D'autres règles sont possibles pour optimiser le processus de réduction.

En ajoutant la règle : (((λx. k p) n_1)n_2 => ((λx. k p) n_1))

<n_1:n_2:$\$_1$, abs(x.@(COMB K,p)),$\$_3$,$\$_4$> ⇒< n_1:$1,abs(x.p),$\$_3$,$4>
(voir la preuve plus en bas, schéma de réduction)

Ainsi la réduction du terme ((λx. λy. x)x)y donne :

1- <[],@(@(abs(x,abs(y,x)),x),y),[],[]> => ^{A0}
   < y:[], @(abs(x,abs(y,x)),x), [],[]>

2- < y:[], @(abs(x,abs(y,x)),x), [],[]> => ^{A0}
   <x:y:[], abs(x,abs(y,x)), [],[]>

3- <x:y:[],abs(x,(abs(y,x)),[],[]>=> ^{A_3}
   <x:y:[],abs(x,(abst (y,x)),[],[] >
   abst(y,x) ->$^{\alpha 2}$ @(COMB K,x)

4- <x:y:[],abs(x, @(COMB K,x)),[],[]> => $^{A}_{\text{nouvelle règle}}$
   <x:[],abs(x,x)), [],[]>

5- <x:[],abs(g_0,x),[],[]>=> ^{A_2}< [],x,[],[]>

A présent on peut donner les règles de transitions finales de la machines

A_0 :<$\$_1$, @(p,q),$\$_3$,$\$_4$> ⇒< q:$\$_1$, p,$\$_3$,$\$_4$>

A_1 :<p:$\$_1$, abs(x,x),$\$_3$,$\$_4$> ⇒< $\$_1$, p,$\$_3$,$\$_4$>

A_2 :<q:$\$_1$, abs(x,p),$\$_3$,$\$_4$> ⇒<$\$_1$, p,$\$_3$,$\$_4$>(p un λ-terme avec x∉ p)

A_3 :<$\$_1$, abs(x,(abs(y,p)),$\$_3$,$\$_4$> ⇒< $\$_1$,abs(x,ABST(y,p)),$\$_3$,$\$_4$>

A_4 :<n_1:$\$_1$, abs(x,@(@(COMB S,p),q)),$\$_3$,$\$_4$>

⇒<(),@(@(@(abs(x,p),n_1),g_0),@(@(abs(x,q),n_1),g_0)),g_0,($\$_1$,$\$_3$):$4>

A_5:<n_1:n_2:$\$_1$, abs(x,@(COMB K,p)),$\$_3$,$\$_4$> ⇒< n_1:$1,abs(x,p),$\$_3$,$4>

A_6: <n_1:$\$_1$, abs(x,@(COMB K,p)),$\$_3$,$\$_4$> ⇒

< n_1, abs(x,p),g_0,($1,$3):$4>

A_7 : <n_1:$\$_1$, abs(x,@(COMB S,p)),$\$_3$,$\$_4$> ⇒

< (), @(@(@(abs(x,p), n_1),g_1), @(g_0,g_1)),g_1:g_0,($\$_1$,$\$_3$):$4>

A_8 : < n:$\$_1$, abs(x,@(p,q)),$\$_3$,$\$_4$> ⇒

<@(abs(x,q),n)):$\$_1$, (@(abs(x,p),n) ,$\$_3$,$\$_4$>

A_9 : < p:$\$_1$, DATA q,$\$_3$,$\$_4$> ⇒ < $\$_1$, p,$\$_3$,m:q:$\$_4$>

A_{10} : < p:$\$_1$, DATA q,$\$_3$,m:$n_1$: $\$_4$> ⇒< $\$_1$,p , $\$_3$, @f($n_1$,DATA q):$\$_4$>

A_{11} : < (), DATA p,$\$_3$,m:$n_1$:$\$_4$> ⇒< (), @f($n_1$,p),$\$_3$,$\$_4$>

A_{12} : < ():$\$_1$,DATA q,v:$\$_3$,():$\$_4$> ⇒ < ():$\$_1$,abs(v,DATA q),$\$_3$,():$\$_4$>

A_{13} : < ():$\$_1$, DATA q,():$\$_3$,(a,v):$\$_4$> ⇒ < a:$\$_1$, DATA q,v,$\$_4$>

A_{14} : < ():$\$_1$,abs(x,@(p,q)),v,$\$_4$> ⇒ < q:$\$_1$,p,x:v:$3,$\$_4$>

A_{15} :< n_1:n_2:n_3:$\$_1$,COMB S,$3,$\$_4$> ⇒

< $\$_1$,abs($n_1$,(abs($n_2$,abs($n_3$,@(($n_1$$n_3$),($n_2$$n_3$))))),$3,$\$_4$>

A_{16}:< n_1:n_2:$\$_1$,COMB K,$3,$\$_4$> ⇒< $\$_1$, n_1,$3,$\$_4$>

< n_1:$\$_1$,COMB I,$3,$\$_4$> ⇒< $\$_1$, n_1,$3,$\$_4$>

A_{17} : < n:$\$_1$,COMB S,$3,$\$_4$> ⇒

< n:$\$_1$, @f(($g_0$$g_2$),($g_1$$g_2$)),$g_2$$g_1$$g_0$:$3,$\$_4$>

A_{18} : < n:$\$_1$, COMB K,$3,$\$_4$> ⇒ < n:$\$_1$, g_0,$g_1$$g_0$:$3,$\$_4$>

A_{19} : < n:$\$_1$, COMB I,$3,$\$_4$> ⇒ < n:$\$_1$, g_0,g_0:$3,$\$_4$>

3.2 Mise en œuvre

Pour la mise en œuvre de l'évaluateur de λ-termes en se basant sur la Cβ+Machine, on a besoin de définir le langage des termes qui seront traités. Ce langage doit contenir tous les termes du λCβ+ calcul puisque la définition de la machine se base sur ce calcul.

3.2.1 Syntaxe du langage

Ce langage contient l'abstraction, l'application et la définition des variables. Les termes sont tous les termes de λCβ+ calcul.

Sa syntaxe est donnée par la grammaire BNF la suivante :

```
<expr> ::=
         VARIABLE
       | LAMBDA  <varlist>.<expr>
       | ( <exprlist> )
       | let <letdef> in <expr>
       | S
       | K
       | I

<letdef>::=
          VARIABLE = <expr>
<varlist>::=
          VARIABLE

        | VARIABLE <varlist>
<exprlist>::=
          <expr>
        | <expr> <exprlist>
```

3.2.2 Schéma de réduction Cb et Cbm^+

Un schéma de réduction est une fonction (Cbm^+), récursive par cas, qui fera correspondre pour chaque terme une étape de réduction. On la définie comme suit

(0)Cbm^+ ((λx. x) N) -> Cbm^+ (N)

(1)Cbm^+ ((λx. y) N) -> Cbm^+ (y) si x ∉y

(2)Cbm^+ ((λx. λy. M) N) -> Cbm^+ ((λx. **cb**$^+$(λy.M)) N)

(a)Cbm^+ ((λx. S P Q) N) -> λg. Cbm^+ ((λx.P)Ng ((λx.Q)Ng)

(b)Cbm^+ ((λx. S P) N) -> λg. λg_1. Cbm^+ ((λx.P)Ng_1 (gg_1))

(c)Cbm^+ ((λx. K P) N) -> λg. Cbm^+ ((λx.P)N)

(d) Cbm^+ ((λx. P Q) N) -> Cbm^+ ((λx.P)N((λx.Q)N)

(A)Cbm^+ (K) -> λg g_1.g

(B)Cbm^+ (I) -> λg .g

(C)Cbm^+ (S) -> λg $g_1g_2.gg_2(g_1g_2)$

avec

Cb$^+$ (λx. x) -> I
Cb$^+$ (λx. y) -> K y
Cb$^+$ ((λx. λy. M) -> **Cb**$^+$ (λx. **Cb**$^+$ (λy.M))
Cb$^+$ ((λx.M N) -> S (λx.M) (λx.N)

Figure 1 : Les règles de réécritures de λCβ$^+$ calcul

Quand plusieurs règles peuvent s'appliquer à une expression, la règle qui figure en premier est la règle qu'il faut appliquer.

3.2.2.1 Optimisation

Le schéma de réduction tel qu'il est défini ci-dessus, peut être optimisé par l'ajout de certaines règles. Ces règles auront pour effet de minimiser les étapes de réductions d'une expressions jusqu'à sa forme normale.

Cbm^+ ((λx. MN) y) = cbm^+(MN) Quand x ∉ (MN) en effet

Cbm^+ ((λx. MN) y) = Cbm^+ (Cbm^+ ((λx. M) y) Cbm^+ ((λx. N) y)) (règle d)

Cbm^+ (((λx. M) y) = Cbm^+(M) (règle 1)

Cbm^+ (((λx. N) y) = Cbm^+(N) (règle 1)

D'où $Cbm^{+}((\lambda x.\ MN)\ y) = cbm^{+}(MN)$ Quand $x \notin (MN)$

$Cbm^{+}((\lambda x.\ Kp)q_1\ q_2) = cbm^{+}((\lambda x.p)\ q_2)$ en effet

$Cbm^{+}((\lambda x.\ Kp)q_1\ q_2) = Cbm^{+}((\lambda g.\ Cbm^{+}\ ((\lambda x.P)q_1))\ q_2\ )$ (règle c)

$Cbm^{+}((\lambda g.Cbm^{+}\ ((\lambda x.P)q_1))\ q_2\ ) = Cbm^{+}((\lambda x.P)q_2\ )$ (règle 1 puisque $g \notin ((\lambda x.P)q_1)$))

D'où $Cbm^{+}((\lambda x.\ Kp)q_1\ q_2) = Cbm^{+}((\lambda x.p)\ q_2)$

La fonction cbm^{+} représente le schéma de réduction de l'évaluateur. Pour la réalisation, on va s'aider des outils d'analyse lex et yacc. Le premier, pour l'analyse lexicale et le second pour l'analyse syntaxique (vérification de la syntaxe des termes). La réduction se fera par la Cβ^{+}Machine qui sera implémenté dans un module à part et avec un langage de programmation fonctionnel. Dans l'annexe, on abordera plus en détail l'implémentation physique.

Conclusion

Dans ce mémoire nous avons présenté une machine à réduction implémentant le système de réécriture de terme le lambda $C\beta^+$ calcul. Notre machine permet de réaliser la réduction forte de lambda terme. En autre le système lambda $c\beta^+$ calcul est un système qui permet de réaliser la réduction de terme pas à pas. Ce système peut constituer un bon outil pédagogique. Il peut être enrichi afin de constituer un utilitaire pour la décision de β équivalence de terme.L'implémentation basée sur la réduction de graphe peut être adaptée à une implémentation parallèle. On pense qu'une telle implémentation va réduire le temps d'exécution d'une façon assez importante.

Bibliographie

Bibliographie

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

[BA78] J.Backus, A Functionnel style and its Algebra of programs Com.of the ACM, vol 21-8,08/78.

[CCM87] G.Cousineau, P.-L.Curien and M. Mauny, The categorical abstract machine. Science of computer programming, 8(2),pp.173-202,1987.

[CF58] H.B.Curry and Feys., Combinatory logic's, *volume 1. Elservier science, Amsterdam,*(1958)

[CGMN80] T. J. W. Clarke, P. J. S. Gladstone, C. D. Maclean, and A. C. Norman. SKIM: The S, K, I reduction machine. In Proceedings of the 1980 Lisp Conference, Stanford, CA, August 1980.

[DB72] N.G.De Bruijn. λ-calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to Church-Rosser theorem. In Indagationes mathematicae, 34, pp 381-392, 1972.

[DOU96] Rémi Douence. Décrire et comparer les mises en œuvres des langages fonctionnels. Thèse Doctorat , Université de renne, Institut de formation en informatique et communication1996.

[GMMS01] M.Gandriau, C.Massoutie,M.Mezghiche,P.Sallé, A calculs without substitution and variable renaming.

[GL02] B.Grégoire, X.Leroy, A compiled Implementation of strong reduction. ICFP'02, October 4-6,2002, Pittsburg, Pennsylvania, USA ACM 1-58113-487-8/02/0010

[HA87] T.Hardin Accart. *Résultats de confluence pour les règles fortes de la logique combinatoire Catégorique et les liens avec les Lambda-Calculs*. thèse de doctorat.27.oct.1987. Université paris VII.

[HAT94] J.Hatcliff and O.Danvy. A generic account of continuation passing style. In Proc.of PoPL'94 pp.458-471,1994.

[HK84] P. Hudak and D. Kranz. A combinator-based compiler for a functional language. In Proceedings 11th ACM Symposium on Principles of Programming languages, pages 122-132, 1984.

[JM82] N. D. Jones and S. S. Muchnick. A fixed-program machine for combinator.

[Joh85] T.Johnsson. Lambda lifting: Transforming programs to recursive equations. In proc.of FPCA'85. LNCS 201,pp.190-203,1985.

[JR84] J.R.Kennaway The complexity of a translation of λ-calculus to combinators Internal Report CSA/13/1984.

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

[KRI96] K.H.Rose. Operational Reduction Models for Functional Programs Languages. Phd thesis, DIKU, Universitetsparken 1, DK-2100 K∅benhaben ∅, February 1996. DIKU report 96/1.

[LAN63] LANDIN,P., The mechanical evaluation of expressions, *Comput.J.*6(1963) 308- 320.

[LAN94] F.LANG. une machine à environnement avec partage, la up-machine. Master's thesis, Université Henri Poincaré, Nancy 1, 1994.

[LAN 98] F .LANG : Un modèle de la β-réduction pour les implantations. Laboratoire de l'informatique du parallélisme. Ecole normale de lyon. 1998

[Le96] X.Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[Lév 80] Lévy, J.J.1980. *Optimal réduction une lambda calculus*. In Essays on Combinatory Logic, pp. 159-92. Hindley et Seldin (éditeurs). Academics press

[LR93] P.Lescanne and J.Rouyer. The calculus of explicite substitutions λν. Internal report, Decembre 1993.

[Pau02] Lawrence C Paulson, Foundations of Functional Programming. Computer Laboratory, University of Cambridge. Computer Science Tripos Part IB Easter Term.

[PJ 90] Simon L. Peyton Jones. Mise en oeuvre des langages fonctionnels de programmation, practice-hall London, Inc.,1990

[RO96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhDthesis, DIKU, Universitetsparken 1, DK-2100 København Ø, February 1996. DIKU report 96/1.

[SOH85] M.J. Shute, P.E. Osmon, and C.L. Hankin. COBWEB: *a combinator réduction engine*. In Proceedings 1985 Conference on Functional Programming

Languages and Computer Architecture, pages 99-112, Nancy, France, 1985.

[Sto85] W. Stoye. *The Implementation of Functional Languages using Custom Hardware.* PhD thesis, University of Cambridge, Computer Laboratory, 1985.

[Sch86] M. Scheevel. NORMA: *a graph reduction processor*. In Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming, Cambridge, Mass,1986.

Annexe

LCβ⁺machine

La machine abstraite Cβ⁺ machine, proposé dans le cadre de ce travail, a été implémenté sous l'environnement MS-DOS, avec le langage de programmation CAML (Categorical Abstract Machine Language) version 0.74[]. La tâche effectuée par Cβ⁺m est l'évaluation d'un programme écrie en λ-cβ⁺calcul.

Conception

Trois modules sont définis pour l'implémentation de Cβ⁺m.

- Le module header qui contient différents types utilisés et toutes les procédures nécéssaires lors de l'analyse du code source.
- Le module cb_red permet de réaliser la réduction.
- Le module main_exec permet d'exécuter un programme à partir d'un fichier ou alors directement reçu sur la console de saisie.

Les types

-Combinateur : Défini le type des combinateur.

- Lambda_terme : c'est un type somme qui contient le type des termes.

La déclaration de ces types en Caml est:

```
type combi =
              I | K | S
 ;;
type terme =
       var of string
     | abs of string * terme
     | app of terme * terme
     | appf of terme * terme
```

Certaines fonctions nécéssaires pour la réduction sont définies.

Fonction abst

C'est la fonction définie pour réaliser une étape d'abstraction, implémente l'algorithme ABST (chapitre 3).

Le code de cette fonction en caml est:

```
let rec abst  = function
  abs (x, var y) ->if (x=y) then ( COMB I)
                              else app( COMB K, var y)
  |  abs(x,abs(y,m1)) ->if (not libre x (abs(y,m1) ))
then  app( COMB K, abs(y,m1))
 else abst(abs (x, abst(abs (y,m1))))
  |  abs(x,app(p,q)) -> if (not libre x (app(p,q)) )
then app( COMB K,app(p,q))
else app(app(COMB S,abs(x,p)),abs(x,q))
;;
```

La fonction libre

C'est une fonction booleane qui permet de vérifier s'il existe des variables libres dans un terme. Elle prend la valeur vraie dans le cas positif, faux dans le cas contraire.

Le code en caml est :

```
let rec libre x = function
     COMB _  -> false
 | var y   -> (x=y)
 | abs (y,t) -> (libre x t ) & (x <> y)
 | app (t1 , t2) -> (libre x t1) or (libre x t2)
;;
```

Fonction réductible

La fonction réductible est définie pour permettre de savoir si un terme peut être réduit. C'est une fonction booleanne, prend la valeur vraie quand le terme peut se réduire et la valeur faux dans le cas contraire.

Le code de la fonction en caml :

```
let rec reductible = function
   app(abs(x,_),_)  -> true
  |app (m1,m2) -> (reductible m1)or(reductible m2)
  | var _        -> false
  | abs(x,m1)  -> reductible m1
  | COMB S       -> true
  | COMB I       -> true
  | COMB K       -> true
```

La fonction print :

Cette fonction est définie pour afficher le résultats de réduction.

Le code de la fonction en caml

```
let rec print_term t =
 match t with
   ([],VAR n,[],[])  -> (print_string n)

 | ([],ABS(n,t),[],[])   -> (print_string ("\\"^n^".");
                  print_term ([],t,[],[]))

 | ([],APP(t1,t2),[],[]) ->
   (print_string "(";print_term ([],t1,[],[]);
 print_string " "; print_term ([],t2,[],[]);
            print_string ")")
;;
```

La fonction gen_var

La fonction gen_var() est définie pour produire d'une façon automatique des variables.
Le code de la fonction en caml est :

```
let var_count = ref 0;;

let gen_var () =
 incr var_count;
 ("g"^(string_of_int !var_count))
;;
```

Analyseur lexical (anal_lex)

S'occupe de la vérification lexicale, reconnaissance des unités lexicales. Si l'expression contient des caractères qui ne peuvent apparaître dans un terme alors une erreur lexicale est renvoyée. Ce module travail avec le module analyseur. Les unités lexicales formées sont envoyées pour l'analyseur, afin de former une unité syntaxique.
Le code de **anal_lex**, est donné ci-dessous.

```
}
rule Token = parse
       [` ` `\t`]        { Token lexbuf }
     | `\n`              { incr lines;
                         words := 0;
                         Token  lexbuf}
     | `\\`              { incr words;
                         LAMBDA  }
     | `.`             { incr words;
                         DOT     }
     | `(`             { incr words;
                         LPAR         }
     | `)`             { incr words;
                         RPAR         }
     | "let"           { incr words;
                         LET     }
     | "in"            { incr words;
                         IN      }
     | "Quit."         { QUIT          }
     | "S"             { incr words;
                         SCOMB        }
     | "K"             { incr words;
                         KCOMB        }
     | "I"             { incr words;
                         ICOMB        }
     | ";;"           { incr words;
                          END     }
     | [`a`-`f``h`-`z`][`a`-`z``A`-`Z``0`-`9``_`]*`\``*
                { let nom = get_lexeme lexbuf in
                         incr words;
                          VARIABLE nom }

     | "(*"     { comment lexbuf}
     | eof      { raise Eof    }
     | _        { raise (LexError (get_lexeme lexbuf)) }

and comment = parse
       "*)"              { Token lexbuf }
     | `\n`              { incr lines;
                         words := 0;
                         Token lexbuf }
     | eof             { raise Eof    }
     | _               { comment lexbuf}
;;
```

Analyseur (anal_syn)

S'occupe de la vérification syntaxique des termes. Si l'expression est mal formulée, une erreur syntaxique est renvoyée.

```
%{
let pile = ref [];;

let depiler () =
 pile := tl !pile ;;

let empiler v =
 pile := v::(!pile);;

let associer v =
 try assoc v !pile with
    Not_found   -> v
  | e       -> raise e
;;

%}

%token LAMBDA DOT  END LET IN LPAR RPAR QUIT SCOMB KCOMB ICOMB
%token <string> VARIABLE
%start command
%type <lcbp__lambda_cl_term> command
%%
command:
          expr  END                       { $1 }
      | QUIT                          { raise EndSession }
;
expr:
          VARIABLE  { (*let v = associer $1 in
                      (VAR v)*) VAR $1  }

      | LAMBDA varlist DOT expr       {
                                          let e =
         list_it (fun v e -> ABS(v,e)) $2 $4 in
         do_list (fun _ -> depiler ()) $2;e }

        | LPAR exprlist RPAR            { it_list (fun f a ->
APP(f,a)) (hd $2) (tl $2) }
        | LET letdef IN expr            { let (v,d) = $2 in
                                          let app =
APP((ABS(v,$4)),d) in
                                          depiler ();
                                          app }
      | SCOMB                         { COMB S }
      | KCOMB                         { COMB K }
      | ICOMB                         { COMB I }
;
letdef:
           VARIABLE EQUAL expr  { empiler ($1,$1); ($1,$3) }
;
varlist:
       VARIABLE          {  empiler ($1,$1 ); [$1]}

     | VARIABLE varlist  {  empiler ($1,$1); $1::$2 }
;
exprlist:
          expr                          { [$1] }
        | expr exprlist                 { $1::$2;  }
;
```

La fonction réduction :

```
let rec term  = function
([],var p,[],[]) -> ([],var p,[],[])
|([],abs(x,p),[],[]) -> if  (reductible p) then term([],p,x::[],[])
                                  else  ([],abs(x,p),[],[])

|(l::r,var y,v,d) -> if ((list_length r = 0)& (not (reductible (l))))
                           then  term ([],app(var y,l),v,d)
                          else (
                                 if (list_length r = 0) then
                                      term([],l,[],(var "$"::[var y],v)::d)
                                  else term (r,l,[],(var "$"::[var y],v)::d)
                                )
|([],var t,[],(var "$"::[var y],v)::(var "$"::r::[],[])::d)->
term([],app(r,app(var y,var t)),[],d)
|([],var t,[],(var "$"::[var y],v)::d)-> term([],app(var y,var t),v,d)

|(p::r, abs(x,var y),v,c )-> if (x=y) then term(r,p,v,c)
                                else term (r,var y,v,c)
|(p::r, abs(x,abs (y,M)),v,d)-> if (not libre x (abs(y, M))) then
term(r,abs(y,M),v,d)
    else term(p::r,(abs(x, abst(abs (y,M) ))),v,d)
|(q::r,abs(x, app(COMB S,t)),v,d )-> let g0 = gen_var() and g1 = gen_var()
in   term(q::var g1::(app(var g0,var g1))::[],abs(x,t),g1::g0::[],(r,v)::d)
|(q::r,abs(x, app(app(COMB S,t),w)),v,d )-> let g0 = gen_var() in
 term(q::(var g0)::(app(app(abs(x,w),q),(var g0)))::[],abs(x,t),g0::[],(r
,v)::d)

|(q::p::l,abs(x, app(COMB K,t)),v,d )-> term(q::l,abs(x,t),v,d)
|(q::r,abs(x, app(COMB K,t)),v,d )-> let g0 = gen_var() in
term(q::[],abs(x,t),g0::[],(r ,v)::d)
|(l::r,abs(x, app(p,q)),v,d )->
        term(l::app(abs(x,q),l)::r,abs(x,p),v,d)
|(l,COMB K,v,d )-> let g0 = gen_var() and g1 = gen_var() in
term(l,abs(g0,abs(g1,var g0)),v,d)
|(r::q::p::l,COMB S,v,d )->
        term(l,app(app(r,p),app(q,p)),v,d)
|(l,COMB S,v,d )-> let g0 = gen_var() and g1 = gen_var() and g2= gen_var()
in term
(l,abs( g0,abs( g1,abs( g2,app(app(var g0,var g2),app(var g1,var g2)
))))),v,d)
```

```
|([],app(p,q),v::r,c)  -> if not reductible (app(p,q)) then
term([],abs(v,app(p,q)),r,c)
                   else term(q::[],p,v::r,c)
|([],app(p,q),[],[])  -> if not reductible (app(p,q)) then
([],app(p,q),[],[]) else term(q::[],p,[],[])


|([],app(p,q),[],(var "$"::[z],[])::(var "$"::r::[],[])::d)  ->
if not reductible (app(p,q)) then term([],app(r,app(z,app(p,q))),[],d)
  else term(q::[],p,[],(var "$"::[z],[])::(var "$"::r::[],[])::d)
|([],app(p,q),[],(var "$"::[r],v)::d)  ->
if not reductible (app(p,q)) then term([],app(r,app(p,q)),v,d)
                           else term(q::[],p,[],(var "$"::[r],v)::d)
|(l,app(p,q),v,c)  -> if not reductible (app(p,q)) then
 term (tl l,hd l,v,(var "$"::[app(p,q)],v)::c)else
                          term (q::l,p,v,c)
|([],p,v,[[],[]])->term([],p,v,[])
|([],p,[],(var "$"::[var y],[])::(var "$"::r::[],[])::d)->
term([],app(r,app(var y,p)),[],d)
|([],p,[],(var "$"::[q],v)::d)->term([],app(q,p),v,d)
|([],p,v,[]) ->  if (list_length v = 1) then ([],abs(hd v,p),[],[]) else
                        term([],abs(hd v,p),tl v,[])
|([],p,[],(c ,v)::d) -> term(c,p,v,d)
|([],p,v::r,d) -> term([],abs(v,p),r,d);;
```