N° d'ordre: 05/91-M/IN

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE
HOUARI BOUMEDIENE (U.S.T.H.B.) ALGER

INSTITUT D'INFORMATIQUE

T H E S E

Pour l'obtention du grade de
MAGISTER EN INFORMATIQUE

PRESENTEE PAR

BENABADJI Karim

V A A L
Langage à Arité Variable
Théorie et Programmation

Soutenue le 29 Juin 1991 à l'Institut d'Informatique (USTHB)

Devant le jury:

Mr A.AINOUCHE	Professeur	U.S.T.H.B.	Président
Mr C.B.BEN-YELLES	Professeur	U.S.T.H.B.	Rapporteur
Mr M.BETTAZ	Professeur	CONSTANTINE	Examinateur
Mme A.ABDERRAHIM	Chargée de cours	U.S.T.H.B.	Examinatrice
Mr M.MEZGHICHE	Chargé de cours	TIZI-OUZOU	Examinateur
Mr K.BELKHIR	Maître-assistant	U.S.T.H.B.	Examinateur

SOMMAIRE

Introduction

Pour la mise en oeuvre des langages fonctionnels, il existe deux principales approches d'implantation efficace:

- La première est basée sur la notion d'environnement, illustrée par l'implantation de ML [Car 83].
- La deuxième repose sur la "machine à réduction" par graphes, technique introduite par Wadsworth [Wad 71], et qui sert de fondement aux implantations de Ponder [Fai 82] et de Lazy ML [Joh 84].

Les langages fonctionnels ont pour avantage majeur leur simplicité sémantique, grâce à un formalisme mathématique qui facilite la compréhension et permet le raisonnement sur les programmes. Par contre, les performances des mises en oeuvres sont altérées, du fait de l'inadéquation avec la machine Von-Neumann.

La disponibilité de la théorie combinatoire TGE basée sur la notion de fonctions à arité variable, nous a incité à définir le langage VAAL (Variable Arity Applicative Language): langage fonctionnel permettant de modéliser les fonctions à arité variable sans artifice de structuration ni de currification. Dans L'univers du système VAAL, les objets sont unifiés: toute fonction est une donnée et toute donnée est une fonction, la différence est strictement contextuelle.

La théorie combinatoire non curryfiée TGE est décrite au chapitre 1.

Dans le chapitre 2. Une description du langage fonctionnel VAAL à arité variable est donnée.

Une première machine à réduction pour VAAL a été conçue et réalisée en se basant sur la réduction par graphe [Bou 90]. Nous proposons maintenant d'implanter une nouvelle machine à réduction, en introduisant une représentation matricielle qui s'inspire de la théorie du Lambda-calcul sur occurrence [Sar 88]. Le processus de réduction devient alors un simple produit matriciel faisant intervenir des matrices creuses.

Dans chapitre 3. nous présentons les algorithmes d'abstractions curryfiés: algorithme d'Abdali, l'algorithme de MaRS et l'algorithme de V.Jau et donnons l'algorithme d'abstraction de VAAL.

Un état de l'art des mises en oeuvres fonctionnelles est présenté dans le chapitre 4.

Finalement, le chapitre 5 est consacré à la machine à réduction matricielle.

Chapitre I: La théorie T_{GE}

Le développement des langages fonctionnels a relancé l'intérêt porté à des théories mathématiques datant des années 30. Parmi celles-ci, la logique combinatoire mise en oeuvre indépendamment par Schöfinkel [Sch 24] en 1924 et Curry [Cur 33] en 1933.

Une des caractéristiques de la logique combinatoire, est que toutes les fonctions sont monaires (currybiées). Aussi la modèlisation d'une fonction n-aire nécessite une étape dite de curryfication qui la transforme en n applications de fonctions unaires.

La théorie T_{GE} [Bel 87, 88] permet pour sa part la modèlisation des fonctions polyadiques, sans l'étape de curryfication, ainsi que celle des fonctions à arité variable. Une fonction à arité variable étant une fonction acceptant un nombre quelconque d'arguments.

A— LE LANGAGE

Cette partie décrit la théorie et rappelle les théorèmes fondamentaux [Bel 87, 88].

1.1 Description

a- Alphabet:

S, K, T, L, D	constantes
V_0, V_1, ...	ensemble dénombrable de variables
$\rightarrow$, =	symboles de réduction et d'égalité
(,)	symboles d'application.

b- T_{GE}-Termes

- Tout atome (constante ou variable) est un terme.
- Si $P_1, \ldots, P_n$ $(n \in \mathbb{N}^*)$ sont des termes alors l'application[1] $(P_1 \ldots P_n)$ est un terme.

c- T_{GE}-formules

Si P et Q sont des termes alors $P \Rightarrow Q$ et $P = Q$ sont des formules.

d- Notations et conventions:

$\equiv$ représente l'identité syntaxique entre deux expressions.

x, y, z, ... lettres minuscules (éventuellement indicées) représentent des variables.

M, N, L, ... lettres majuscules (éventuellement indicées) représentent des termes.

i, j, k, l, m, n, ... représentent des entiers naturels.

$\in$ représente l'appartenance ensembliste (ou l'occurrence).

$\mathbb{N}$ est l'ensemble des entiers naturels.

$\mathbb{N}^* = \mathbb{N} - \{0\}$

'...' indique l'utilisation d'expressions d'indices consécutifs croissants. Ainsi: $X_1 \ldots X_4 \equiv X_1\ X_2\ X_3\ X_4$

$X_1 \ldots X_n \equiv$ séquence formée des n termes $X_1, \ldots, X_n$ $(n \in \mathbb{N}^*)$.

---

(1) Bellot a utilisé les notations $P_1 : P_2 \ldots P_n$ et $P_1(P_2, \ldots, P_n)$, faisant implicitement la distinction entre la tête P_1 et le reste $P_2 \ldots P_n$ qui est nommé séquence.

1.2 T_{GE} : La théorie

a- T_{GE}-Axiomes

La théorie T_{GE} possède les schémas d'axiomes suivants, dont les T_{GE}-axiomes sont des instances:

$((K\ X_1 \ldots X_n)\ Y_1 \ldots Y_m) \Rightarrow X_1$

$((S\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow ((G_1\ X_1 \ldots X_n)(G_2\ X_1 \ldots X_n) \ldots (G_m\ X_1 \ldots X_n))$

$((T\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow ((G_1\ X_1 \ldots X_n)\ X_2 \ldots X_n)$

$((L\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow ((G_1\ X_1 \ldots X_n)\ (G_2\ X_1 \ldots X_n)\ X_1 \ldots X_n)$

$((D\ G_1 \ldots G_m)\ X) \Rightarrow (G_1\ X)$

$((D\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow (G_2\ X_1\ X_2 \ldots X_n)$

Remarques: - La réduction associée à D dépend de la taille de la 2ème séquence d'arguments (atomique ou composée).

- La notation utilisée pour énoncer les axiomes fait apparaître implicitement des conditions sur le nombre d'arguments.

b- T_{GE}-Règles

i) Réduction: La réduction en T_{GE} est définie par les règles d'inférence suivantes(2):

$$(a)\ \frac{X_1 \Rightarrow Y_1 \ \ldots\ X_n \Rightarrow Y_n}{(F\ X_1 \ldots X_n) \Rightarrow (F\ Y_1 \ldots Y_n)} \qquad (b)\ \frac{F \Rightarrow G}{(F\ X_1 \ldots X_n) \Rightarrow (G\ X_1 \ldots X_n)}$$

$$(t)\ \frac{M \Rightarrow N \qquad N \Rightarrow L}{M \Rightarrow L} \qquad (r)\ \frac{}{M \Rightarrow M}$$

ii) Egalité: L'égalité est la relation d'équivalence engendrée par la réduction. Elle est définie au moyen des règles (a), (b), (t) et (r), auxquelles on ajoute:

$$(e)\ \frac{M \Rightarrow N}{M = N} \qquad (s)\ \frac{M = N}{N = M} \qquad (t')\ \frac{M = N \qquad N = L}{M = L}$$

Exemple: la famille de combinateurs $(P_k)_{k\geq 1}$ (appelés sélecteurs) définie comme suit:

$$P_1 \equiv (S\ K\ K)$$

$$P_{k+1} \equiv (T\ (K\ P_k))$$

a la propriété suivante: $(P_k\ X_1 \ldots X_n) \Rightarrow X_k$ Si $1 \leq k \leq n$

---

(2) Les règles sont données à la manière de Hindley [Hin 86]. c'est à dire prémisse(s) de la règle au dessus de la barre horizontale et conclusion au dessous.

1.3 Définitions:

a- Sous-terme

La relation P est un sous-terme de Q est définie inductivement comme suit :

a) P est un sous-terme de P.

b) Si P est un sous-terme de M_1 et/ou, ..., et/ou de M_n alors P est un sous-terme de $(M_1 \ldots M_n)$

b- Combinateur

Un terme ne contenant aucune variable comme sous-terme est appelé combinateur.

c- Redex et Contractum

Dans un axiome, La partie gauche est appelée Redex, et la partie droite est appelée Contractum.

d- Garbage

On appelle garbage les termes de la forme :

$$((S\ F)\ X_1 \ldots X_n)$$

$$((T\ G_1 \ldots G_m)\ X)$$

$$((L\ F)\ X_1 \ldots X_n)$$

$$((D\ G)\ X_1 \ldots X_n)$$

e- Forme normale

Un terme ne contenant ni redex ni garbage comme sous-terme est dit en forme normale.

f- Terme normalisable

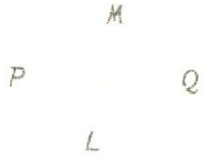
Un terme M égal à une forme normale N est dit normalisable, et N est sa forme normale.

g- L'ordre normal:

La réduction en ordre normal consiste à réduire le redex le plus à gauche et le plus externe.

1.4 Lemme du Losange:

Si $M \Rightarrow P$ et $M \Rightarrow Q$, il existe L tel que $P \Rightarrow L$ et $Q \Rightarrow L$



1.5 Théorèmes de Church-Rosser:

a- Premier théorème de Church-Rosser (CR1):

Si $M = N$, il existe L tel que $M \Rightarrow L$ et $N \Rightarrow L$.

b- Deuxième théorème de Church-Rosser (CR2):

Si $M \Rightarrow N$ et si N est en forme normale, alors il existe une réduction de M vers N suivant l'ordre normal.

Les propriétés suivantes sont déduites de (CR1) et (CR2).

1.6 Corollaires:

- Si M = N et N est en forme normale, alors M => N.
- La forme normale d'un terme est unique (si elle existe).

1.7 Théorème de Consistance:

T_{GE} est une théorie consistante.

B— L'ABSTRACTION DANS T_{GE}

Dans les théories de combinateurs, étant donné un terme E contenant éventuellement une variable x, il existe, un terme F ne contenant aucune occurrence de x, tel que: (F x) = E. En fait, plusieurs algorithmes dits d'abstractions permettent de déterminer de tels termes F.

Dans les théories currybiées, l'abstraction d'un terme E par rapport à plusieurs variables $x_1, \ldots, x_n$, est obtenu en procédant à n abstractions par rapport à $x_1, \ldots, x_n$ successivement.

La théorie T_{GE} n'étant pas currybiée, permet de définir une abstraction par rapport à plusieurs variables $x_1, \ldots, x_n$ parallélement.

1.8 Définition : Substitution

Soit $k \in \mathbb{N}^*$, la substitution simultanée de x_1 par $N_1, \ldots, x_k$ par N_k dans M, notée $[N_1 \ldots N_k/x_1 \ldots x_k]M$, est définie inductivement comme suit :

- $[N_1 \ldots N_k/x_1 \ldots x_k]x_i \equiv N_i \qquad (1 \leq i \leq k)$
- $[N_1 \ldots N_k/x_1 \ldots x_k]c \equiv c$, si $c \notin \{x_1, \ldots, x_k\}$
- $[N_1 \ldots N_k/x_1 \ldots x_k](F\ y_1 \ldots y_n) \equiv$

$([N_1 \ldots N_k/x_1 \ldots x_k]F\ [N_1 \ldots N_k/x_1 \ldots x_k]y_1 \ \ldots \ [N_1 \ldots N_k/x_1 \ldots x_k]y_n)$

1.9 Théorème de complétude:

Soient M un terme, $k \in \mathbb{N}^*$ et $x_1, \ldots, x_k$ des variables, il existe un terme noté $[\backslash x_1 \ldots x_k.M]$ ne contenant aucune des variables $x_1, \ldots, x_k$ tel que :

$$([\backslash x_1 \ldots x_k.M]\ N_1 \ldots N_k) \Rightarrow [N_1 \ldots N_k/x_1 \ldots x_k]M$$

Preuve : Etant donnés M, x_1, ..., x_k, on peut construire un tel terme en utilisant l'algorithme (d'abstraction) suivant:

- $[\backslash x_1 \ldots x_k.x_i] \equiv P_i$
- $[\backslash x_1 \ldots x_k.y] \equiv (K\ y)$, si $y \notin \{x_1, \ldots, x_k\}$
- $[\backslash x_1 \ldots x_k.(F\ M_1 \ldots M_n)] \equiv$

$$(S\ [\backslash x_1 \ldots x_k.F][\backslash x_1 \ldots x_k.M_1] \ldots [\backslash x_1 \ldots x_k.M_n])$$

Cette notion d'abstraction, étant donnée une famille de terme $(E_k)_{k \geq 1}$, permet de trouver une solution F_k, pour chaque $k \in \mathbb{N}^*$, de l'équation $(f\ x_1 \ldots x_k) = E_k$.

Bellot s'est aussi intéressé à une nouvelle notion d'abstraction en relation avec la notion d'arité variable introduite dans T_{GE} [BEL 87], dans le cas où la famille de

terme E_k répond à certaines formes de définitions réccurentes. Dans ce cas le problème consiste à déterminer un terme F, solution du système infini d'équations $\{\& x_1 \ldots x_k = E_k \ / \ k \in \mathbb{N}^*\}$ et non pas une famille de termes [Jay 89].

Afin de manipuler un nombre infini d'équations, nous allons introduire les notions de métaterme et métaséquence qui simulent la famille de termes à partir d'une forme générale (définition récurrente).

1.10 Les métatermes et métaséquences: (3)

L'idée de base est de représenter à l'aide de l'expression $x_1..x_\delta$. la famille de séquences de méta-variables d'indices consécutifs croissants $\{x_1 \ldots x_n \ / \ n \in \mathbb{N}^*\}$. On utilisera le symbole δ pour faire référence à la dernière méta-variable.

Les métatermes et métaséquences en $x_1..x_\delta$ sont définis inductivement:

- Tout atome (constante ou variable) est un 1-métaterme(4).
- x_i est un i-métaterme avec $i \in \mathbb{N}^*$.
- $x_{\delta-j}$ est un $j+1$-métaterme avec $j \in \mathbb{N}$.

---

(3) Les métatermes (resp. métaséquences) sont appelés dans [Bel 88] A-termes (resp. A-séquences).

(4) A chaque métaterme (resp. métaséquence), est associée une borne inférieure pour la valeur des tailles de séquences d'arguments admissibles.

Preuve:

Un tel terme est donné par l'algorithme d'abstraction générale.

Soient a *un atome*

$F, t, M_1, \ldots, M_k$ *des métatermes*

s *une métaséquence*

et $i,\ p \in \mathbb{N}^*$, $q \in \mathbb{N}$

- $\Omega[x_i] \equiv P_i$
- $\Omega[x_{\delta - i}] \equiv Pd_i$
- $\Omega[a] \equiv (K\ a)$
- $\Omega[(F\ x_p..x_{\delta-q})] \equiv (A\ \Omega[(W_{p,q}\ F)]$
- $\Omega[(F\ M_1 \ldots M_k)] \equiv (S\ \Omega[F]\ \Omega[M_1] \ldots \Omega[M_k])$
- $\Omega[(F\ s\ t)] \equiv \Omega[((U\ F\ t)\ s)]$
- $\Omega[(F\ s\ x_p..x_{\delta-q})] \equiv (A\ \Omega[((V_{p,q}\ F)\ s)])$

Les combinateurs A, U, P_i, Pd_i, $W_{p,q}$, $V_{p,q}$ *utilisés dans l'algorithme admettent les réductions suivantes avec* $(n, i, p \in \mathbb{N}^*)$ *et* $(q \in \mathbb{N})$:

$((A\ F)\ X_1 \ldots X_n) \Rightarrow ((F\ X_1 \ldots X_n)\ X_1 \ldots X_n)$

$((U\ F\ M)\ X_1 \ldots X_n) \Rightarrow (F\ X_1 \ldots X_n\ M)$

$(P_i\ X_1 \ldots X_n) \Rightarrow X_i$ $(n \geq i)$

$(Pd_i\ X_1 \ldots X_n) \Rightarrow X_{n-i}$ $(n \geq i)$

$((W_{p,q}\ F)\ X_1 \ldots X_n) \Rightarrow (F\ X_p \ldots X_{n-q})$ $(p+q \leq m)$

$(((V_{p,q}\ F)\ X_1 \ldots X_n)\ Y_1 \ldots Y_m) \Rightarrow (F\ X_1 \ldots X_n\ Y_p \ldots Y_{m-q})$ $(p+q \leq m)$

Exemple: La fonction $+_2$ *fait la somme du 2ème à l'avant dernier argument.*

$+_2 \equiv \Omega[(+\ x_2..x_{\delta-1})] \equiv (A\ \Omega[(W_{2,1}\ +)])$

$\equiv (A\ (S\ \Omega[W_{2,1}]\ \Omega[+])) \equiv (A\ (S\ (K\ W_{2,1})\ (K\ +)))$

Chapitre II: Le langage VAAL

Le premier langage d'inspiration fonctionnelle Lisp [McC 62] est né dans les années 60. Ce langage a donné naissance à de nombreux dialectes très utilisés, tels que Common-Lisp [Ste 84] et Le-Lisp [Cha 84]. Il faut noter que ces langages intègrent pour la plupart des opérations d'effet de bord et par conséquent, ne sont pas purement fonctionnels.

L'introduction du langage FP [Bac 78] a suscité un regain d'intérêt pour cette famille de langages. De nouveaux langages ont été proposés comme, ML [Mil 84], MIRANDA [Tur 85] ou GRAAL [Bel 86] qui se caractérisent par un temps de réponse rapide. Leurs implantations sont prometteuses et seront d'autant plus rapide avec de nouvelles architectures.

2.1 La notion d'arité variable dans Lisp

Le langage Lisp permet d'intégrer des fonctions à arité variable comme l'addition généralisée Plus telle que:

$Plus(x_1,\ldots,x_n) = x_1+\ldots+x_n$ quelque soit $n \in \mathbb{N}^*$ et définie comme suit:

```
(Def Plus (l)
     (COND (= l NIL) 0
           (TRUE (+ (CAR l) (Plus (CDR l))))  ; )
```

Lisp gère l'arité variable en organisant les arguments dans une liste l. En fait Lisp simule l'arité variable au moyen d'une structure de données(5).

Notre objectif, en proposant le langage fonctionnel VAAL (Variable Arity Applicative Language), est d'utiliser directement les fonctions à arité variable sans simulation (sans l'utilisation d'une structure visible au programmeur), en se basant sur la théorie TGE.

2.2 VAAL-expression

Les objets de base de VAAL, nommés atomes sont:

- les identificateurs, qui permettent de nommer les fonctions.
- les nombres (entiers ou décimaux).

On construit l'ensemble des VAAL-expressions symboliques (en abrégé VAAL-expression) par induction:

* tout atome est une VAAL-expression;
* toute suite finie de VAAL-expressions séparées par des espaces et délimitées par des parenthèses est une VAAL-expression.

---

(5)FP utilise une structure de séquence délimitée par des crochets.

Remarque: Les VAAL-expressions se représentent par une notation préfixée totalement parenthèsée: la fonction précède ses arguments, et une paire de parentheses permet de déterminer le début et la fin de l'appel de la fonction. Les arguments sont soit des atomes, soit de nouveau des expressions parenthèsées.

2.3 Evaluateur VAAL

Une notion importante rattachée aux systèmes de programmation est l'interaction. Un système de programmation est interactif lorsque le calcul est effectué dès son énonciation.
Lorsqu'un système de programmation applicative est interactif, l'introduction d'une expression au clavier est immédiatement suivie de l'impression du résultat de son évaluation . Ce comportement du système se définit plus précisément ainsi [Sai 91]:
Une boucle d'interaction est un enchaînement potentiellement infini de trois opérations effectuées par un système de programmation applicative:

- la lecture de l'expression à évaluer,
- l'évaluation de celle-ci et,
- l'impression du résultat.

La boucle d'interaction du système de programmation VAAL manifeste son attente d'une expression à évaluer, en émettant sur le terminal un prompteur (un point d'interrogation). Le résultat de l'évaluation est imprimé précédé du signe "=". Ensuite, à moins que l'expression précédente n'ait demandé l'arrêt du système, le prompteur apparait de nouveau.

2.4 Les fonctions arithmétiques:

Ce sont les fonctions arithmétiques "+", "*", "-" et "/" à arité variable.

Une définition récurrente informelle de chacune de ces fonctions de VAAL (OP) est obtenue à partir de la fonction binaire correspondante (opb) de la manière suivante:

(OP x) = x

(OP x_1 ... x_n) = (opb (OP x_1 ... x_{n-1}) x_n) $n \geq 1$

Exemples : i) ? (+ 2)
= 2

ii) ? (* 2 3)
= 6

iii) ? (- 10 5 2)
= 3

iv) ? (/ 120 15 4 2)
= 1

De la même manière que les fonctions arithmétiques, on définit les fonctions logiques et les fonctions relations à arité variable.

2.5 Les fonctions relations:

Ce sont les fonctions relations "=" "<>" "<", "≤", ">", "≥" et "ATOM" (qui teste si ses arguments sont atomiques) à arité variable.

Une définition récurrente informelle de ces fonctions relations (PR) est obtenue à partir de la fonction (prf) d'arité fixe de la manière suivante:

Si prf est d'arité 1

(PR $x_1 \ldots x_n$) = (prf x_1) et (prf x_2) et...et (prf x_n) $n \geq 1$

Si prf est d'arité 2

(PR x) = (prf x x)

(PR $x_1 \ldots x_n$) = (prf x_1 x_2) et (prf x_2 x_3) et...et (prf x_{n-1} x_n) $n \geq 2$

Ces fonctions retournent un résultat booléen représenté par "TRUE" pour Vrai et par "NIL" pour Faux.

Exemples : i) ? (= (+ 1 2 2) (- 10 5) 5)
= TRUE

ii) ? (2 10 4 25 1)
= NIL

2.6 Les fonctions logiques:

Ce sont les fonctions logiques "NOT", "OR" et "AND" à arité variable.

(NOT $\langle v_1 \rangle \ldots \langle v_n \rangle$) : évalue successivement les expressions $\langle v_1 \rangle \ldots \langle v_n \rangle$, et retourne NIL dès qu'une évaluation délivre TRUE, sinon TRUE.

(OR $\langle v_1 \rangle \ldots \langle v_n \rangle$) : évalue successivement les différentes expressions $\langle v_1 \rangle \ldots \langle v_n \rangle$ jusqu'à ce que l'une de ces évaluations ait une valeur TRUE, auquel cas OR retourne la valeur TRUE sinon OR retourne NIL. Si aucune expression n'est fournie, cette fonction retourne NIL.

(AND $\langle v_1 \rangle \ldots \langle v_n \rangle$) : évalue successivement les différentes expressions $\langle v_1 \rangle \ldots \langle v_n \rangle$ jusqu'à ce que la valeur d'une évaluation soit égale à NIL, à ce moment AND retourne NIL sinon AND retourne la valeur TRUE. Si aucune expression n'est fournie, cette fonction retourne TRUE.

Exemples: i) ? (AND (= 1 1) (< 2 3 4))
= TRUE

ii) ? (OR (= 2 2) (< 2 3 4) (< 1 2 0))
= TRUE

iii) ? (NOT (= 2 2) (> 4 3 2) (< 1 2 0))
= NIL

2.7 La conditionnelle:

Sa forme générale est la suivante:

(IF ($\langle c_1 \rangle$ $\langle v_1 \rangle$)
($\langle c_2 \rangle$ $\langle v_2 \rangle$)
..
($\langle c_k \rangle$ $\langle v_k \rangle$)
$\langle v_{k+1} \rangle$)

Les arguments de IF sont des listes (appelées clauses) (⟨c_i⟩ ⟨v_i⟩) où ⟨c_i⟩ et ⟨v_i⟩ sont des VAAL-expressions représentant respectivement la condition à tester et l'expression à évaluer.

Les clauses sont étudiées séquentiellement à partir de la première clause en évaluant ⟨c_i⟩:

- Si cette évaluation retourne la valeur TRUE, les autres arguments de IF sont ignorés, et l'expression ⟨v_i⟩ correspondante est évaluée.

- Si l'évaluation du test retourne NIL, le reste de la clause est ignoré, et l'analyse recommence à partir de la clause suivante.

Si tous les tests ont échoué, ⟨v_{k+1}⟩ est évaluée.

La conditionnelle permet donc d'effectuer éventuellement plusieurs tests à la suite, en provoquant l'évaluation de la VAAL-expression ⟨v_i⟩ associée au premier test ⟨c_i⟩ réussi.

Exemples:

```
? (IF ((= 1 1) 'premier)
      ((= 2 1) 'deuxième)
               'le-reste)
= premier
```

```
? (IF ((> 1 1) 'premier)
      ((= 2 1) 'deuxième)
               'le-reste)
= le-reste
```

2.8 La fonction d'inhibition:

Dans VAAL, toute VAAL-expression est susceptible d'une évaluation immédiate. Pour introduire des données, il est donc nécessaire d'empêcher son évaluation. L'introduction de la fonction 'QUOTE' (notée aussi " ' ") permet de signaler à l'évaluateur que son argument est à considérer littéralement, sans effectuer la moindre analyse.

```
Exemples : ? '(+ 2 8)
           = (+ 2 8)

           ? 'a
           = a
```

2.9 Les fonctions de manipulation des expressions parenthèsées:

Les expressions parenthèsées sont des constructions et constituent une structure de données dénommée liste, qui peut être destructurée afin d'exhiber ses composantes nommées champs. L'expression parenthésée a seulement deux champs:

* le premier élément d'une liste correspond au premier champ;
* le reste de la liste (la liste privée de son premier élément) correspond au deuxième champ.

Les deux fonctions qui permettent d'accéder à ces champs se nomment CAR et CDR et sont à arité variable.

```
Exemples : i) ? (CAR '(1 2))
              = 1

           ii) ? (CAR '(1 2) '(3 4))
               = (1 3)

          iii) ? (CDR '(1 2 3))
               = (2 3)

           iv) ? (CDR '(1 2 3) '(a z e))
               = ((2 3) (z e))
```

La fonction qui permet de construire une expression parenthèsée est nommée CONS, elle retourne en valeur une liste.

Une définition récurrente informelle de la fonction CONS est obtenue à partir de la fonction binaire 'cons' de la manière suivante:

(CONS $\langle v_1 \rangle \ldots \langle v_n \rangle$) $\equiv$ (cons (CONS $\langle v_1 \rangle \ldots \langle v_{n-1} \rangle$) $\langle v_n \rangle$)

```
Exemples : ? '(+ 2 8)
           = (+ 2 8)

           ? 'a
           = a
```

2.9 Les fonctions de manipulation des expressions parenthèsées:

Les expressions parenthèsées sont des constructions et constituent une structure de données dénommée liste, qui peut être destructurée afin d'exhiber ses composantes nommées champs. L'expression parenthésée a seulement deux champs:

* le premier élément d'une liste correspond au premier champ;
* le reste de la liste (la liste privée de son premier élément) correspond au deuxième champ.

Les deux fonctions qui permettent d'accéder à ces champs se nomment CAR et CDR et sont à arité variable.

```
Exemples : i) ? (CAR '(1 2))
              = 1

           ii) ? (CAR '(1 2) '(3 4))
               = (1 3)

          iii) ? (CDR '(1 2 3))
               = (2 3)

           iv) ? (CDR '(1 2 3) '(a z e))
               = ((2 3) (z e))
```

La fonction qui permet de construire une expression parenthèsée est nommée CONS, elle retourne en valeur une liste.

Une définition récurrente informelle de la fonction CONS est obtenue à partir de la fonction binaire 'cons' de la manière suivante:

(CONS <v_1>...<v_n>) = (cons (CONS <v_1>...<v_{n-1}>) <v_n>)

Exemples : i) ? (CONS 1 '())
= (1)

ii) ? (CONS 5 (CDR '(2 3)) '(7 8))
= ((5 3) 7 8)

Remarques: * Nous confondons volontairement la liste vide et NIL.

* La notion de liste permet de retourner plusieurs résultats, en fait une liste de résultats que l'on peut exploiter par application de CAR et CDR.

2.10 La fonction évaluateur:

La forme générale de Apply est:

(APPLY ⟨f⟩ ⟨v1⟩...⟨vn⟩) où

⟨f⟩, ⟨v1⟩,...,⟨vn⟩ sont des VAAL-expressions.

Son premier argument ⟨f⟩ est appliqué aux arguments ⟨vi⟩.

Exemples: ? (APPLY + 1 2 4)
= 7

? (APPLY CAR '(1 2) '(3 4))
= (1 3)

? (APPLY (CAR '(/ 120 15 4 2)) 10 5 2)
= 1

Remarque: Le langage VAAL est méta-récursif: les programmes se confondent avec les données.

En effet, la VAAL-expression (/ 120 15 4 2) est considérée comme un programme; et dans l'expression ci-dessus, la même VAAL-expression est apparue comme donnée.

Cette unification entre programme et donnée possède les avantages suivants:

- considérer une donnée comme programme permet d'écrire rapidement d'autres systèmes de programmation.
- considérer un programme comme donnee permet de soumettre son programme à un programme d'analyse de programmes, qui va tenter d'en démontrer des propriétés et d'en proposer des améliorations.

Cette facilité de changement de statut explique pourquoi les outils d'aide à la mise au point sont souvent écrits dans des langages méta-récursifs.

2.11 Définition d'une fonction utilisateur.

Afin de pouvoir manipuler dans une définition de fonction les arguments "dont le nombre n'est pas préalablement fixé", on introduit les notions suivantes :

2.11.1 VAAL-Séquence:

Une VAAL-séquence est une paire d'entiers naturels préfixés par un signe.

$$[(+/-)i,(+/-)j] \quad \text{où } i,j \in N.$$

Etant donnée une séquence quelconque d'arguments, $X_1 \ldots X_n$, une VAAL-séquence représente une suite finie d'arguments d'indice consécutifs croissants ou décroissants.

Ainsi Si $i \geq 0$ et $j \geq 0$ et $n \geq i+j$ Alors

[+i,+j] dénote la sous-séquence d'arguments $X_i \ldots X_j$
si $i<j$ elle est d'indices croissants sinon décroissants.

[-i,-j] dénote la sous-séquence d'arguments $X_{n-i} \ldots X_{n-j}$
si $i>j$ elle est d'indices croissants sinon décroissants.

[+i,-j] dénote la sous-séquence d'arguments $X_i \ldots X_{n-j}$ d'indices croissants.

[-i,+j] dénote la sous-séquence d'arguments $X_{n-i} \ldots X_j$ d'indices décroissants.

Remarques: - La longueur de la séquence sera définie lors de l'appel.

- Les VAAL-séquences représentent les métatermes de T_{GE}.

Exemples: Par rapport à une séquence à 5 arguments X_1 X_2 X_3 X_4 X_5
[+1,-0] représente la séquence d'arguments d'indices croissants allant du premier au dernier argument X_1 X_2 X_3 X_4 X_5.
[-0,+1] représente la séquence d'arguments d'indices décroissants allant du dernier au premier argument X_5 X_4 X_3 X_2 X_1.
[+2,-1] représente la séquence d'arguments d'indices croissants allant du deuxième à l'avant dernier argument X_2 X_3 X_4.

2.11.2 Le sélecteur:

c'est une fonction qui sélectionne un argument dans une séquence de longueur indéfinie, noté:

[(-,+,i) où i > 0

Exemples: [-i] : sélectionne le $i^{ème}$ argument à partir du dernier

[-0] : sélectionne le dernier argument.

[+i] : sélectionne le $i^{ème}$ argument.

2.11.3 La def-application:

(ndf VAAL-[illegible] ...) ou (ndf sélecteur)

où ndf est un identificateur de fonction définie au préalable.

2.11.4 Définition de fonction:

Le système VAAL permet à l'utilisateur d'enrichir son ensemble de fonctions calculables en l'autorisant à définir ses propres fonctions à arité variable:

(DEF ndf (l_{min}) ⟨f⟩ ⟨g⟩) où

ndf est l'identificateur de la fonction.

l_{min} est le nombre minimal d'arguments de la fonction.

⟨f⟩ est une VAAL-expression non evaluable correspondant au cas l_{min}.

⟨g⟩ est une VAAL-expression non évaluable correspondant au cas général.

Exemples:

```
i) ? (DEF Cube(2)                    ;calcule le cube de
        ()                           ;l'avant dernier argument
        (* [-1] [-1] [-1])

   = CUBE

ii) ? (DEF Max(1)                                    ;calcule le maximum
        ([+1])                                       ;d'une séquence
        (IF ((2 [+1] [-1]) (MAX [+1,-1]))            ;d'arguments en
                           (MAX [+2,-0]))            ;comparant les 2
        ))                                           ;extrémité

   = MAX
```

2.12 L'application:

Elle permet de lancer le processus d'exécution dans VAAL avec une syntaxe conforme à la notation Polonaise préfixée. Ainsi elle assure l'homogénéité des écritures traduisant l'appel à l'évaluateur VAAL.

Sa syntaxe est la suivante :

$(ndb\ \langle v_1\rangle \ldots \langle v_m\rangle)$ où

ndb est l'identificateur d'une fonction définie dont le nombre minimal d'argument est l_{min} ($n \geq l_{min}$).

$\langle v_i\rangle$ est la ième forme représentant l'argument effectif de l'appel.

Exemples: i) ? (CUBE 6 3 5 2 1)

= 8

ii) ? (MAX 9 10 5 6 7 3 5 6)

= 10

2.13 Exemples de fonction VAAL:

(i) PGCD: la fonction qui calcule le PGCD.

```
? (DEF PGCD(:)
    ([+1])
    (IF ( (= (% [+1] [+2]) 0) (PGCD [+2,-0]) )
                              (PGCD [+2,-0] (% [+1] [+2])))
    ))

= PGCD
```

(% est la fonction qui calcule le reste d'une division entière)

```
? (PGCD 75 45 25)

= 5
```

(ii) Mfact: la fonction $(x_1*x_2*...x_n)+...+(x_1*x_2*x_3)+(x_1*x_2)+x_1$

```
? (DEF Mfact(1)
    ([+1])
    (+ (* [+1.-0]) (Mfact [+1,-1])))

= Mfact

? (Mfact 2 3 6)

= 44.
```

(iii) APPEND. la fonction de concaténation des listes fournies en arguments.

```
? (DEF Append (:)
    (IF ( (ATOM [+1]) 'NIL) [+1])
    (IF ( (ATOM [+1]) (APPEND [+2,-0]) )
                      (CONS (CAR [+1]) (APPEND (CDR [+1]) [+2,-0])))
    ))

= APPEND

? (APPEND '(1 2 3) '(4 5 6) '(8 9 0))

= (1 2 3 4 5 6 8 9 0)
```

VAAL est un langage fonctionnel qui puise ses fondements dans la théorie TGE et manipule les fonctions à arité variable sans artifice de structuration et ni de curryfication.

La mise en oeuvre de VAAL est basé sur la théorie combinatoire TGE et la tâche de manipulation des arguments est pris en charge par les combinateurs définis dans TGE.

L'algorithme d'Abdali et de MaPS introduisent la notion d'abstraction multiple simultanée: l'abstraction se fait par rapport à la séquence $x_1 \ldots x_n$.

L'algorithme de V.Jay s'intéresse au cas où le "terme" E correspond à la famille de termes récurrents $\{E_n \ , (n \geq 1)\}$. L'abstraction produit alors un "combinateur" F correspondant à la famille de combinateurs récurrents $\{F_n = [x_1 \ldots x_n]E_n \ , (n \geq 1)\}$ tels que $(F_n \ x_1 \ \ldots \ x_n) = E_n$.

Nous avons vu au chapitre I que la théorie non-curryfiée T_{CE} permet de définir, une abstraction variable intégrant l'abstraction simple et l'abstraction multiple simultanée.

B— ABSTRACTION CURRYFIEE

3.1 Algorithme d'Abdali [Abd 76]

Cet algorithme permet l'abstraction par rapport à plusieurs variables simultanément.

3.1.1 Règles de réduction

La réduction est définie au moyen des quatres combinateurs I, K_n, $I_{n,m}$ et $B_{n,m}$:

$I \ x \Rightarrow x$

$K_n \ x_1 x_2 \ldots x_n \Rightarrow x_1$

$I_{n,m} \ x_1 \ldots x_n \Rightarrow x_m \qquad (n \geq m \geq 1)$

$B_{n,m} \ x_1 x_2 \ldots x_n \ y_1 \ldots y_n \Rightarrow (x_1 \ (x_2 \ y_1 \ldots y_n) \ldots (x_n \ y_1 \ldots y_n))$

3.1.2 Définitions

Etant donnée une expression combinatoire E,

* Si E ≡ AB alors A est la composante immédiate gauche de E et B est la composante immédiate droite de E

* Une composante de E est, soit E lui-même, soit une composante d'une composante immédiate de E.

* Une composante initiale de E est soit E lui-même, soit une composante initiale d'une composante immédiate gauche de E.

* Une composante primaire de E est soit une composante immédiate droite d'une composante initiale de E, soit une composante initiale de E qui est un atome.

Exemple: Soit E ≡ SK (x (KK) yz) (S (az) (SSy)) (xyz)

Composantes initiales de E:

S, SK, SK (x (KK) yz), SK (x (KK) yz) (S (az) (SSy)), E.

Composante primaires de E: S, K, x (KK) yz, S (az) (SSy), xyz

Remarque: Les composantes primaires de E correspondent aux sous-termes de E au premier niveau et les composantes initiales de E retracent le parenthèsage implicite à gauche de l'application.

3.1.3 Algorithme d'Abstraction

Etant donnés E un terme combinatoire et $x_1,\ldots,x_n$ des variables, alors l'abstraction de E par rapport à $x_1,\ldots,x_n$ est le terme noté $[x_1\ldots x_n]E$ défini par:

1. $[x_1\ldots x_n]E \equiv K_n\ E$ $\qquad x_i \notin E,\ 1\le i\le n$
2. $[x_1\ldots x_n]x_1\ldots x_n \equiv I$
3. $[x_1\ldots x_n]x_i \equiv I_{n,i}$ $\qquad 1\le i\le n$
4. $[x_1\ldots x_n]g\ x_1\ldots x_n \equiv g$ $\qquad x_i \notin g,\ 1\le i\le n$
5. $[x_1\ldots x_n]g\ x_m\ldots x_n \equiv [x_1\ldots x_{m-1}]\ g$ $\qquad x_n \notin g,\ m\le i\le n$
6. $[x_1\ldots x_n]x_i\ b_2\ldots b_m \equiv B_{n,n}\ I\ I_{n,i}\ ([x_1\ldots x_n]b_2)\ldots([x_1\ldots x_n]b_m)$
   où $b_2,\ldots,b_m$ sont des composantes primaires de E et $1\le i\le n$
7. $[x_1\ldots x_n]\ b_1\ldots b_m \equiv B_{n,n-1}b_1([x_1\ldots x_n]b_2)\ldots([x_1\ldots x_n]b_m)$
   où b_1 est la plus longue composante initiale de E ne contenant pas x_i $(1\le i\le n)$ et $b_2,\ldots,b_m$ sont des composantes primaires de E

Exemple: L'abstraction de $E = SK\ (x\ (KK)\ yz)\ (S\ (az)\ (SSy))\ (xyz)$ par rapport à x,y,z.

$B_{3,3}(SK)([x\ y\ z](x(KK)yz))([x\ y\ z](S(az)(SSy)))([x\ y\ z](xyz))$ {7}

$\equiv B_{3,3}(SK)([x]x(KK))([x\ y\ z](S(az)(SSy)))([x\ y\ z](xyz))$ {5}

$\equiv B_{3,3}(SK)([x]x(KK))(B_{3,2}S([x\ y\ z](az))([x\ y\ z](xyz))$ {7}

$\equiv B_{3,3}(SK)([x]x(KK))(B_{3,2}S([x\ y\ z](az))([x\ y\ z](SSy)))I$ {2}

$\equiv B_{3,3}(SK)(B_{1,2}I_{1,1}([x](KK)))(B_{3,2}S([x\ y\ z](az))([x\ y\ z](SSy)))I$ {6}

$\equiv B_{3,3}(SK)(B_{1,2}I_{1,1}([x](KK)))(B_{3,2}S([x,y]a)([x,y,z](SSy)))I$ {8}

$\equiv B_{3,3}(SK)(B_{1,2}I_{1,1}([x](KK)))(B_{3,2}S([x\ y]a)(B_{3,1}(SS)([x\ y\ z]y)))I$ {7}

$\equiv B_{3,3}(SK)(B_{1,2}I_{1,1}(K_1(KK)))(B_{3,2}S([x\ y]a)(B_{3,1}(SS)([x\ y\ z]y)))I$ {1}

$\equiv B_{3,3}(SK)(B_{1,2}I_{1,1}(K_1(KK)))(B_{3,2}S(K_2a)(B_{3,1}(SS)([x\ y\ z]y)))I$ {1}

$\equiv B_{3,3}(SK)(B_{1,2}I_{1,1}(K_1(KK)))(B_{3,2}S(K_2a)(B_{2,1}(SS)I_{3,2}))I$ {3}

3.2 La machine MaRS [MaR 87]

La machine MaRS utilise un langage fonctionnel dont elle exploite le parallélisme intrinsèque.

L'algorithme de MaRS est dicté par la forme du terme à traiter.

3.2.1 Règles de réduction

Soient les variables x et y, et les expressions $e_1,\ldots,e_n$, les combinateurs sont I, B, M_n, W_n, N_n, Q_n, K_n définis par:

$I\ x \Rightarrow x$ (identité)

$B\ x\ y\ z \Rightarrow x\ (y\ z)$ (composition)

$M_n\ e_1 \ldots e_n \Rightarrow e_1\ e_n\ e_2 \ldots e_{n-1}$ (déplacement) $n\geq 3$

$W_n\ e_1 \ldots e_n \Rightarrow e_1\ e_n\ e_2 \ldots e_{n-1}\ e_n$ (copie et déplacement) $n\geq 2$

$K_n\ e_1 \ldots e_n \Rightarrow e_1\ c_2 \ldots e_{n-1}$ (destruction) $n\geq 2$

$N_n\ e_1 \ldots e_n \Rightarrow e_1\ (e_2\ e_n)\ e_3 \ldots e_{n-1}$ (création de noeud) $n\geq 3$

$Q_n\ e_1 \ldots e_n \Rightarrow e_1\ (e_2\ e_n)\ e_3 \ldots e_n$ (copie et création de noeud) $n\geq 3$

On remarque que les indices des combinateurs indiquent l'élément qui sera déplacé, copié ou détruit. Les combinateurs habituels C et K correspondent respectivement à N_3 et K_2.

3.2.2 L'algorithme d'abstraction

Avant de donner l'expression combinatoire finale, l'algorithme d'abstraction de MaRS, donne une expression intermédiaire équivalente appelé forme canonique.

i) l'expression sous forme canonique:

Le principe de l'algorithme est de réduire par étapes successives l'expression E en une forme $(k\ x_1 \ldots x_n)$, où k est une combinaison de combinateurs et de constantes de E. Après

chaque étape de transformation, l'expression intermédiaire équivalente à E possède la forme canonique suivante:
$(a\ e_1 \ldots e_p\ v_1 \ldots v_m)$, $p \geq 0$, $0 \leq m \leq n$ où a est une constante, les e_i sont des expressions telles que e_1 n'est pas une constante, $v_1 \ldots v_m$ est un sous ensemble ordonné de $x_1 \ldots x_n$.

(0.1) $[x_1 \ldots x_n]x_i\ e_2 \ldots e_p\ v_1 \ldots v_m \equiv I\ x_i\ e_2 \ldots e_p\ v_1 \ldots v_m$

(0.2) $[x_1 \ldots x_n]k_1 \ldots k_\ell\ e_1 \ldots e_p\ v_1 \ldots v_m \equiv (k_1 \ldots k_\ell)e_1 \ldots e_p\ v_1 \ldots v_m$

(1) $[x_1 \ldots x_n]k\ x_i\ e_2 \ldots e_p\ v_1 \ldots v_m \equiv M_j\ k\ e_2 \ldots e_p\ u_1 \ldots u_{m+1}$
$x_i \notin (v_1 \ldots v_m)$ et $u_1 \ldots u_{m+1}$ est obtenu à partir de $v_1 \ldots v_m$ dans lequel x_i a été inséré, de telle manière que $u_1 \ldots u_{m+1}$ reste un sous ensemble ordonné de $x_1 \ldots x_n$. L'index j est défini par $j = p+q$, où q est tel que $x_i \equiv u_q$

(2) $[x_1 \ldots x_n]k\ x_i\ e_2 \ldots e_p\ v_1 \ldots v_m \equiv W_j\ k\ e_2 \ldots e_p\ v_1 \ldots v_m$
avec $j = p+q$, où q est tel que $x_i \equiv v_q$ et $x_i \in (v_1 \ldots v_m)$

(3) $[x_1 \ldots x_n]k\ (d_1 \ldots d_r\ x_i)\ e_2 \ldots e_p\ v_1 \ldots v_m \equiv$

$N_j\ k\ (d_1 \ldots d_r)\ e_2 \ldots e_p\ u_1 \ldots u_{m+1}$

avec $j = p+q+1$, $x_i \notin (v_1 \ldots v_m)$ et $r \geq 1$, où $u_1 \ldots u_{m+1}$ et q sont définis comme dans la règle (2)

(4) $[x_1 \ldots x_n]k\ (d_1 \ldots d_r\ x_i)\ e_2 \ldots e_p\ v_1 \ldots v_m \equiv$

$Q_j\ k\ (d_1 \ldots d_r)\ e_2 \ldots e_p\ v_1 \ldots v_m$

avec $j = p+q+1$, $x_i \notin (v_1 \ldots v_m)$, $r \geq 1$, où q est tel que $x_i \equiv v_q$

(5) $[x_1 \ldots x_n]k\ (d_1 \ldots d_r)\ e_2 \ldots e_p\ v_1 \ldots v_m \equiv$

$B\ k\ (d_1 \ldots d_{r-1})\ d_r\ e_2 \ldots e_p\ v_1 \ldots v_m$

avec $r > 1$ et $d_r \neq x_i$

ii) l'expression combinatoire finale

L'expression initiale se trouve transformée en une expression de la forme $a\ v_1 \ldots v_m$ $(p=0)$. Après quoi, afin d'extraire les variables de l'expression, on utilise les règles suivantes, autant de fois que nécessaire:

(6) Si $x_n \equiv v_m$, $n \geq 1$

$$[x_1 \ldots x_n](k\ v_1 \ldots v_m) \equiv [x_1 \ldots x_{n-1}](k\ v_1 \ldots v_{m-1})$$

(7) Si $x_n \not\equiv v_m$, $m \geq 0$

$$[x_1 \ldots x_n](k\ v_1 \ldots v_m) \equiv [x_1 \ldots x_{n-1}](K_j\ k\ v_1 \ldots v_{m-1})$$

avec $j = m+2$

(8) $[]k \equiv k$

Exemple: Abstraction de l'expression $(a\ (b\ (b\ x\ y)\ (b\ x\ y)))$ où a et b sont des constantes par rapport à b, x, et y:

$[b\ x\ y]a(b\ (b\ x\ y)\ (b\ x\ y))$

$\equiv B\ a\ (b\ (b\ x\ y))\ (b\ x\ y))$ (5)

$\equiv (B\ a)\ (b\ (b\ x\ y))\ (b\ x\ y))$ (0.2)

$\equiv (B\ (B\ a))\ b\ (b\ x\ y)\ (b\ x\ y)$ (5 et 0.2)

$\equiv (M_4\ (B\ (B\ a)))\ (b\ x\ y)\ (b\ x\ y)\ b$ (1 et 0.2)

$\equiv (N_5\ (M_4\ (B\ (B\ a))))\ (b\ x)\ (b\ x\ y)\ b\ y$ (3 et 0.2)

$\equiv (N_5\ (N_5\ (M_4\ (B\ (B\ a))))\ b)\ (b\ x\ y)\ b\ x\ y$ (3 et 0.2)

$\equiv (Q_5\ (N_5\ (N_5\ (M_4\ (B\ (B\ a))))\ b))\ (b\ x)\ b\ x\ y$ (4 et 0.2)

$\equiv (Q_4\ (Q_5\ (N_5\ (N_5\ (M_4\ (B\ (B\ a))))\ b)))\ b\ b\ x\ y$ (4 et 0.2)

$\equiv (W_2\ (Q_4\ (Q_5\ (N_5\ (N_5\ (M_4\ (B\ (B\ a))))\ b))))\ b\ x\ y$ (2 et 0.2)

$\equiv (W_2\ (Q_4\ (Q_5\ (N_5\ (N_5\ (M_4\ (B\ (B\ a))))\ b))))$ (6)

3.3 Algorithme de V.Jay [Jay 89]

L'algorithme de V.Jay ne s'intéresse plus un terme mais à une famille de termes récurrents.

3.3.1 Notion de séquence

Une séquence est une juxtaposition de variables, notée (x,n) ou $x_1..x_n$ où n est "un entier indéterminé". Les ".." dénotent l'énumération formelle d'objets, en l'occurrence des variables, indicés dans l'ordre croissant, l'indice s'incrémentant de 1 à chaque pas.

Une sous-séquence de $x_1..x_n$ est un terme $x_a..x_b$ où a et b sont des expressions arithmétiques dépendant exclusivement de n. Par exemple, on peut écrire $x_i..x_{n-j}$ où $i \in \mathbb{N}^*$ et $j \in \mathbb{N}$.

3.3.2 Règle de réduction

On retrouve les combinateurs récurrents K_n, S'_n, B_n, Q_n, E_n et C_n.

$S'_n\ f\ g\ x_1...x_n \Rightarrow f\ x_1...x_n\ (g\ x_1...x_n)$

$K_n\ f\ x_1...x_n \Rightarrow f$

$B_n\ f\ g\ x_1...x_n \Rightarrow f\ (g\ x_1...x_n)$

$Q_n\ f\ x_1...x_n\ y \Rightarrow f\ y\ x_1...x_n$

$C_n\ f\ y\ x_1...x_n \Rightarrow f\ x_1...x_n\ y$

$E_n\ f\ y\ x_1...x_n \Rightarrow f\ (...(f\ (f\ y\ x_1)\ x_2)...)\ x_n$

L'algorithme de V.Jay s'énonce en deux parties d'une part l'abstraction par rapport à une variable, et d'autre part l'abstraction par rapport aux séquences qui fera l'objet du paragraphe suivant.

3.3.3 Abstraction par rapport à une variable

Cet algorithme est une généralisation de celui de la logique Combinatoire.

1. $[x]\ x \equiv K_0$
2. $[x]\ T \equiv K_1\ T$ avec $x \notin T$
3. $[x](M\ y_a..y_b) \equiv E_{b-a+1}\ C_1\ ([x]M)\ y_a..y_b$
4. $[x](M\ N) \equiv S'_1\ ([x]M)\ ([x]N)$

3.3.4 Abstraction par rapport à une séquence de variables

Faire l'abstraction sur une séquence d'arguments non spécifiée $x_1..x_n$ dans un terme se réalise en appliquant récursivement les règles suivantes:

soient $1 \le a \le b \le n$ et $1 \le c \le d \le m$

5. $[x_1..x_n]T \equiv K_n\ T$ avec $x_i \notin T$
6. $[x_1..x_n]x_a \equiv K_{a-1}\ K_{n-a}$
7. $[x_1..x_n](x_a..x_b) \equiv K_{a-1}\ (E_{b-a+1}\ K_{n-b}\ T)$
8. $[x_1..x_n](M\ x_a..x_b) \equiv S'_n\ (Q_n\ (B_1\ K_{a-1}\ (B_{b-a+1}\ K_{n-b}))))\ [x_1..x_n]M$
9. $[x_1..x_n](M\ y_c..y_d) \equiv E_{d-c+1}\ C_n\ ([x_1..x_n]M)\ y_c..y_d$
10. $[x_1..x_n](M\ L) \equiv S'_n\ ([x_1..x_n]M)\ ([x_1..x_n]L)$
11. $[x_1..x_n]x_1..x_n \equiv K_0$
12. $[x_1..x_n]M\ x_1..x_n \equiv M$ si $x_i \notin M$
    $\equiv S'_n\ (Q_n\ K_0)\ ([x_1..x_n]M)$ sinon
13. $[x_1..x_n](M\ x_a..x_b) \equiv B_1\ K_{a-1}\ (B_{b-a+1}\ K_{n-b})\ M$ si $x_i \notin M$
14. $[x_1..x_n](M\ y_c..y_d) \equiv K_n\ (M\ y_c..y_d)$ si $x_i \notin M$

Exemple: Abstraction de l'expression ($\delta\ x_1..x_n\ (\delta\ y)$) par rapport à δ, $x_1..x_n$ et y revient à $[\delta]\ ([x_1..x_n]\ ([y]\ (\delta\ x_1..x_n\ (\delta\ y))))$

* Abstraction par rapport à y:

$[y](\delta\ x_1..x_n\ (\delta\ y))$

$\equiv S'_1\ ([y]\delta\ x_1..x_n)\ ([y]\delta\ y)$ (4)

$\equiv S'_1\ (K_1\ (\delta\ x_1..x_n))\ (S'_1\ (K_1\ \delta)\ K_0)$ (2 et 1)

$\equiv B_1\ (\delta\ x_1..x_n)\ \delta$ (Opt)

* Abstraction par rapport à $x_1..x_n$:

$[x_1..x_n]\ (B_1\ (\delta\ x_1..x_n)\ \delta)$

$\equiv S'_n\ ([x_1..x_n]B_1\ (\delta\ x_1..x_n))\ ([x_1..x_n]\delta)$ (10)

$\equiv S'_n\ (S'_n\ (K_n\ B_1)\ ([x_1..x_n](\delta\ x_1..x_n)))\ (K_n\ \delta)$ (10 et 5)

$\equiv S'_n\ (S'_n\ (K_n\ B_1)\ \delta)\ (K_n\ \delta)$ (12)

$\equiv C_n\ (B_n\ B_1\ \delta)\ \delta$ (Opt)

* Abstraction par rapport à δ:

$[\delta]C_n\ (B_n\ B_1\ \delta)\ \delta$

$\equiv S'_1\ ([\delta]C_n\ (B_n\ B_1\ \delta))\ [\delta]\delta$ (4)

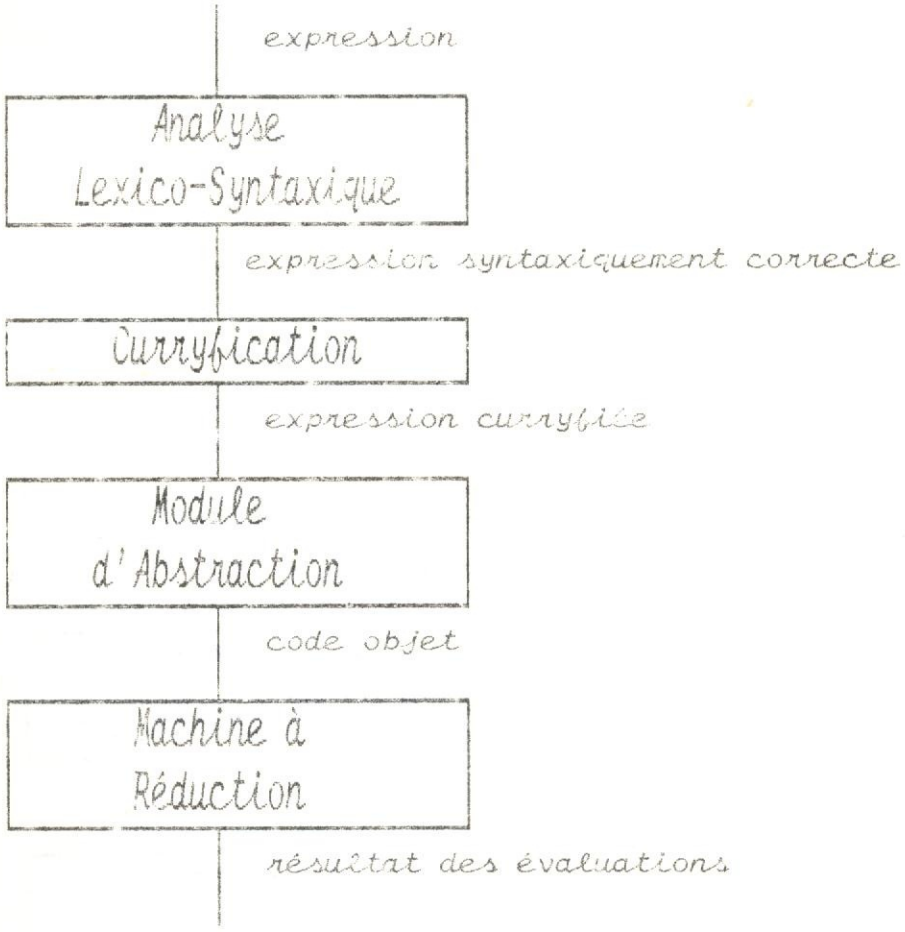
$\equiv S'_1\ (S'_1\ ([\delta]C_n)\ ([\delta](B_n\ B_1\ \delta)))\ I$ (4 et 1)

$\equiv S'_1\ (S'_1\ (K_1\ C_n)\ (B_n\ B_1))\ I$ (2 et Opt)

$\equiv S'_1\ (B_1\ C_n\ (B_n\ B_1))\ I$ (Opt)

Remarque: L'algorithme présenté ci-dessus donne le moyen effectif de réaliser l'abstraction d'une séquence de variables de longueur indéterminée dans un terme, et un deuxième algorithme que l'on ne présentera pas donne les règles d'abstraction d'une variable dans un terme [Jay 89].

Les algorithmes d'abstractions que nous avons présentés s'appliquent à des fonction curryfiées. Les interprétations des langages fonctionnels classiques basés sur ces théories répond au schéma suivant:

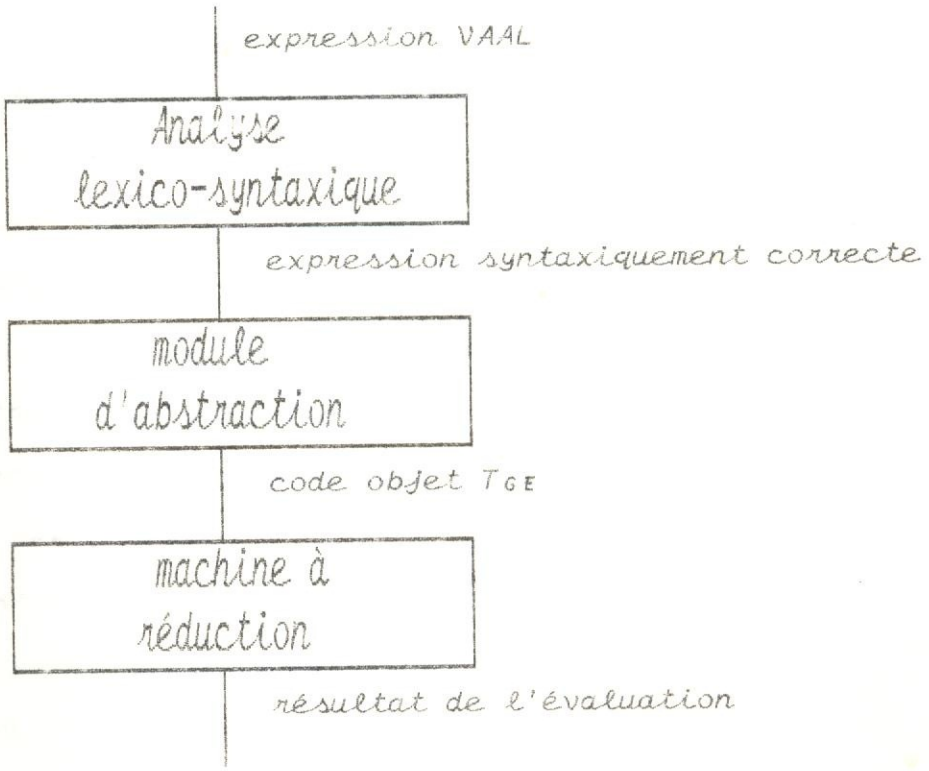


A- INTERPRETEUR DE LANGAGE FONCTIONNEL CLASSIQUE

L'approche T_{GE}, nous permettra de compiler (faire l'abstraction) une expression VAAL en utilisant des combinateurs non curryfiées.

B- ABSTRACTION NON CURRYFIEE

La modèlisation des fonctions à arité variable se fait dans TGE au moyen d'un algorithme d'abstraction qui manipule les notions de métatermes et métaséquences définis au chapitre I. Nous utiliserons pour déterminer la représentation d'une fonction à arité variable dans le langage VAAL. L'étape de curryfication n'a plus lieu d'exister et l'interprétation de VAAL répond au schéma suivant:



B- INTERPRETEUR DU LANGAGE VAAL

3.4 Algorithme d'abstraction de VAAL

Nous ferons une analogie entre les VAAL-séquences et les sélecteurs définis dans VAAL, et les métatermes et métaséquences définis dans T_{GE}.

3.4.1 VAAL-termes et VAAL-séquences:

De même que dans T_{GE}, on définit pour VAAL, les notions de VAAL-terme et VAAL-séquence de la manière suivante :

- Tout atome (constante ou variable) est un $VAAL_1$-terme.
- Tout sélecteur $[+i]$ est un $VAAL_i$-terme avec $i \geq 1$.
- Tout sélecteur $[-i]$ est un $VAAL_{i+1}$-terme avec $i \geq 0$.
- Tout $VAAL_k$-terme est une $VAAL_k$-séquence.
- $[+i,-j]$ est une $VAAL_{i+j}$-séquence avec $i \geq 1$ et $j \geq 0$
- $[+i,+j]$ est une $VAAL_j$-séquence avec $i < j$ et $i \geq 1$ et $j \geq 1$
- $[-i,-j]$ est une $VAAL_{i+1}$-séquence avec $i > j$ et $i \geq 0$ et $j \geq 0$
- $[-i,+j]$ est une $VAAL_{i+j}$-séquence avec $i \geq 0$ et $j \geq 1$
- $[+i,+j]$ est une $VAAL_i$-séquence avec $i > j$ et $i \geq 1$ et $j \geq 1$
- $[-i,-j]$ est une $VAAL_j$-séquence avec $i < j$ et $i \geq 0$ et $j \geq 0$
- Si s est une $VAAL_p$-séquence et t un $VAAL_q$-terme alors $s\ t$ est une $VAAL_r$-séquence avec $r = Max(p,q)$.
- Si s est une $VAAL_p$-séquence alors $s\ [(+/-)i,(+/-)j]$ est une $VAAL_r$-séquence avec $r = Max(p, i+j)$.
- Si b est un $VAAL_p$-terme et s une $VAAL_q$-séquence alors $(b\ s)$ est une $VAAL_r$-séquence avec $r = Max(p,q)$.

Alors que les classes des métatermes et métaséquences définies dans TGE [Bel 87] ne manipulent que des suites d'indices croissants, les classes des VAAL-termes et VAAL-séquences representent aussi bien des séquences croissantes que décroissantes.

3.4.2 L'algorithme d'abstraction

Etant donnée une VAAL-expression M, l'abstraction (optimisée) de M, notée $\Omega[M]$, est définie par:

Case M of DEF l_{min} 6 9)	:: (Demin $\Omega[6]$ $\Omega[9]$)	
a	:: (K a)	(a constante)
[+i]	:: P_i	
[-i]	:: Pd_i	
(F [+i,+j])	:: (A $\Omega[(Z_{i,j}\ F)]$)	(i<j)
(F [-i,-j])	:: (A $\Omega[(Y_{i,j}\ F)]$)	(i>j)
(a [+1,-0])	:: a	(a constante)
(F [+1,-0])	:: (A $\Omega[F]$)	
(F [+i,-j])	:: (A $\Omega[(W_{i,j}\ F)]$)	
(a [-0,+1])	:: (G a)	(a constante)
(F [-0,+1])	:: (Ad $\Omega[F]$)	
(F [-i,+j])	:: (A $\Omega[(X_{i,j}\ F)]$)	
(F [+i,+j])	:: (Ad $\Omega[(Y_{i-1,j-1}\ F)]$)	(i>j)
(F [-i,-j])	:: (Ad $\Omega[(Z_{i+1,j+1}\ F)]$)	(i<j)
(F s [+i,-j])	:: (A $\Omega[((V_{i,j}\ 6)\ s)]$)	
(F s [-i,+j])	:: (A $\Omega[((R_{i,j}\ 6)\ s)]$)	
(F s [+i,+j])	:: (A $\Omega[((O_{i,j}\ 6)\ s)]$)	(i<j)
(F s [-i,-j])	:: (A $\Omega[((Q_{i,j}\ 6)\ s)]$)	(i>j)

$(F\ s\ [+i,+j])$ :: $(Ad\ \Omega[((Q_{i-1,j-1}\ b)\ s)])$ $(i>j)$

$(F\ s\ [-i,-j])$ :: $(Ad\ \Omega[((O_{i+1,j+1}\ b)\ s)])$ $(i<j)$

$(M_1 \ldots M_j)$ :: $(K\ (M_1 \ldots M_j))$ (M_i constantes)

$(F\ M_1 \ldots M_j)$ :: $(C\ \Omega[F]\ M_1 \ldots M_j)$ (M_i constantes)

$(a\ M_1 \ldots M_j)$ :: $(B\ a\ \Omega[M_1] \ldots \Omega[M_j])$ (a constante)

$(F\ M_1 \ldots M_j)$ :: $(S\ \Omega[F]\ \Omega[M_1] \ldots \Omega[M_j])$

$(F\ s\ t)$ :: $\Omega[((U\ F\ t)\ s)]$

$(IF\ M_1\ M_2 \ldots M_{j-1}\ M_j)$ :: $(IF\ \Omega[M_1]\ \Omega[M_2] \ldots \Omega[M_{j-1}]\ \Omega[M_j])$

où les combinateurs introduits sont tels que:

$((A\ F)\ x_1 \ldots x_n)$ => $((F\ x_1 \ldots x_n)\ x_1 \ldots x_n)$

$((Ad\ F)\ x_1 \ldots x_n)$ => $((F\ x_1 \ldots x_n)\ x_n \ldots x_1)$

$((Z_{i,j}\ F)\ x_1 \ldots x_n)$ => $(F\ x_i \ldots x_j)$

$((Y_{i,j}\ F)\ x_1 \ldots x_n)$ => $(F\ x_{n-i} \ldots x_{n-j})$

$((X_{i,j}\ F)\ x_1 \ldots x_n)$ => $(F\ x_{n-i} \ldots x_j)$

$((W_{i,j}\ F)\ x_1 \ldots x_n)$ => $(F\ x_i \ldots x_{n-j})$

$(((V_{i,j}\ F)\ x_1 \ldots x_n)\ y_1 \ldots y_m)$ => $(F\ x_1 \ldots x_n\ y_i \ldots y_{m-j})$

$(((R_{i,j}\ F)\ x_1 \ldots x_n)\ y_1 \ldots y_m)$ => $(F\ x_1 \ldots x_n\ y_{m-i} \ldots y_j)$

$(((Q_{i,j}\ F)\ x_1 \ldots x_n)\ y_1 \ldots y_m)$ => $(F\ x_1 \ldots x_n\ y_{m-i} \ldots y_{m-j})$

$(((O_{i,j}\ F)\ x_1 \ldots x_n)\ y_1 \ldots y_m)$ => $(F\ x_1 \ldots x_n\ y_i \ldots y_j)$

$(P_i\ x_1 \ldots x_n)$ => x_i

$(Pd_i\ x_1 \ldots x_n)$ => x_{n-i}

$((D_i\ g_1 \ldots g_m)\ x_1 \ldots x_i)$ => $(g_1\ x_1 \ldots x_i)$

$((D_i\ g_1 \ldots g_m)\ x_1 \ldots x_i \ldots x_n)$ => $(g_2\ x_1 \ldots x_i \ldots x_n)$

$((B\ g_1 \ldots g_m)\ x_1 \ldots x_n)$ => $(g_1\ (g_2\ x_1 \ldots x_n)\ \ldots\ (g_m\ x_1 \ldots x_n)$

$((C\ g_1 \ldots g_m)\ x_1 \ldots x_n)$ => $((g_1\ x_1 \ldots x_n)\ g_2 \ldots g_m)$

$((G\ F)\ x_1 \ldots x_n)$ => $(F\ x_n \ldots x_1)$

$((U\ F\ G)\ x_1 \ldots x_n)$ => $(F\ x_1 \ldots x_n\ G)$

Exemples: La fonction Max qui calcule le maximum d'une séquence d'arguments en comparant les arguments situés aux deux extrémités d'une séquence d'arguments est définie par:

```
(DEF Max (1)
    ([+1])
    (IF ((≥ [+1] [-0]) (MAX [+1,-1]))
                       (MAX [+2,-0]) ))
```

En appliquant l'algorithme d'abstraction, on obtient:

```
(D P1 (IF (B ≥ P1 Pd0) (A (K (W1,1 Max)))
          (K true) (A (K (W2,0 Max)))))
```

Pour conclure, nous allons comparer l'algorithme d'abstraction de V.Jay et celui de VAAL sur l'exemple concret de la fonction Mfact définie comme suit:

```
(Def Mfact (1)
    ([+1])
    (+ (* [+1,-0]) (Mfact [+1,-1])))
```

* Abstraction de l'algorithme de V.Jay

```
(IF (= 1 n) I (S'n (Bn + *) ((B1 K0 (Bn-1 K1)) Mfact)))
```

* Abstraction de VAAL

```
(D P1 (B + (A (K (W1,1 Mfact)))))
```

On remarque que:

- L'abstraction de V.Jay porte sur une expression curryfiée alors que celle de VAAL porte sur une expression non curryfiée.

- Les résultats obtenus par l'algorithme d'abstraction de V.Jay dépendent d'itérateurs contrairement à l'algorithme de VAAL qui produit un code unique.

- Le code obtenu pour l'algorithme de VAAL est plus compact que celui de V.Jay et devrait induire une machine à réduction plus rapide.

- L'implantation de l'algorithme d'abstraction de VAAL est uniforme alors que celui de V.Jay, nécessite une identification de la nature de l'abstraction (une variable ou une séquence de variables).

INTRODUCTION

Les langages fonctionnels puisent leurs bases théoriques dans le Lambda-calcul [Chu 41], la Logique combinatoire [Cur 33] et [Sch 24] ou des théories équivalentes. Ainsi le langage Lisp est une approximation de la théorie Lambda-calcul [Per 78] et les systèmes FP de Backus [Rob 80] ont pour modèle la Logique combinatoire. Il est à noter que dans ces théories, toutes les fonctions sont monaires (curryfiées). Or la plupart des langages fonctionnels permettent la manipulation et des fonctions polyadiques (arité fixe) et des fonctions à arité variable (nombre quelconque d'arguments).

La gestion de telles fonctions est ramenée, dans tous les langages fonctionnels usuels, à la gestion de fonctions monaires. Les arguments sont organisés dans une structure de données (en général la liste) qui permet de gérer l'arité variable.
En Lisp, la fonction 'Plus' à arité variable est écrite:

```
(DEF Plus (l)
     (COND  ((= l '()) 0)
            (TRUE (+ (CAR l) (Plus (CDR l)))))))
```

En fait, nous voyons que dans ce cas Lisp simule l'arité variable au moyen de la liste l (alors que FP utilise une structure de séquence délimitée par des crochets).

Chapitre IV: Mises en oeuvres des langages fonctionnels

Le développement des langages fonctionnels a longtemps été compromis par l'inefficacité des implantations réalisées [Pey 82].

Suite aux actives recherches [Bac 78] [Pey 84] dont ont fait l'objet les langages fonctionnels, la situation a maintenant évoluée et des implantations rivalisant en efficacité avec des compilateurs de langages plus conventionnels [Pey 82] sont proposés tels que ML de Cardelli [Car 83], Ponder de Fairbairn [Fai 82], et le compilateur de Lazy ML de Chalmers [Joh 84].

Nous présentons ici les deux principales approches d'implantation des langages fonctionnels:

- La première est basée sur la notion d'environnement, illustrée par l'implantation de ML, d'inspiration Lispienne.
- La deuxième repose sur la réduction par graphes, technique introduite par Wadsworth [Wad 71], et qui a servi de fondement aux implantations de Ponder et de Lazy ML.

A- LES MACHINES A ENVIRONNEMENT

La mise en oeuvre des machines à environnement peut être vue comme une implantation du lambda-calcul, dans la mesure où le principe de calcul repose sur les règles de réduction définies dans le lambda-calcul.

4.1 Lisp et son implantation [McC 62][All 78]

L'implantation de Lisp fut l'une des premières mises en oeuvre d'un langage fonctionnel.

La définition d'une fonction Lisp est effectuée en utilisant un formalisme appelé "notation lambda", qui permet d'associer un nom à une expression. L'association d'un nom à une lambda-expression définit une fonction dont les paramètres correspondent aux variables liées de la lambda-expression. Ainsi, la fonction: produit(x,y) = x*y peut être définie par (def produit (lambda(x y)(* x y))), expression directement traduite de l'écriture: \xy.x*y.

Dans la β-réduction, la liaison d'une variable correspond à la substitution de toutes les occurrences de la variable par l'argument correspondant, à l'intérieur du corps de la fonction. En Lisp, cette substitution n'est jamais effectuée directement mais remplacée par un mécanisme équivalent: pour une stratégie par valeur, l'argument est évalué avant l'appel, et la valeur est conservée durant toute l'évaluation de la fonction. La "substitution" est réalisée à chaque utilisation du paramètre correspondant. Cela permet en particulier d'éviter un renommage systématique de variables.

Le mécanisme de base pour la substitution consiste à regrouper les paramètres dans un objet appelé environnement, et à remplacer toute substitution par un accès à cet objet. L'environnement contient l'ensemble des variables accessibles à un moment donné ainsi que leurs valeurs courantes.

4.1 Lisp et son implantation [McC 62][All 78]

L'implantation de Lisp fut l'une des premières mises en oeuvre d'un langage fonctionnel.

La définition d'une fonction Lisp est effectuée en utilisant un formalisme appelé "notation lambda", qui permet d'associer un nom à une expression. L'association d'un nom à une lambda-expression définit une fonction dont les paramètres correspondent aux variables liées de la lambda-expression. Ainsi, la fonction: produit(x,y) = x*y peut être définie par (def produit (lambda(x y)(* x y))), expression directement traduite de l'écriture: \xy.x*y.

Dans la β-réduction, la liaison d'une variable correspond à la substitution de toutes les occurrences de la variable par l'argument correspondant, à l'intérieur du corps de la fonction. En Lisp, cette substitution n'est jamais effectuée directement mais remplacée par un mécanisme équivalent: pour une stratégie par valeur, l'argument est évalué avant l'appel, et la valeur est conservée durant toute l'évaluation de la fonction. La "substitution" est réalisée à chaque utilisation du paramètre correspondant. Cela permet en particulier d'éviter un renommage systématique de variables.

Le mécanisme de base pour la substitution consiste à regrouper les paramètres dans un objet appelé environnement, et à remplacer toute substitution par un accès à cet objet. L'environnement contient l'ensemble des variables accessibles à un moment donné ainsi que leurs valeurs courantes.

Que se passe-t-il lorsqu'une variable libre apparaît dans une fonction? Elle peut être liée dans l'environnement de définition de cette fonction (langage à portée lexicale ou statique) ou dans un environnement d'exécution (langage à portée dynamique).

La plupart des interprètes Lisp actuels implantent une portée dynamique. En effet, ils utilisent une simplification qui consiste à dire que la valeur d'une abstraction (\x.N) est elle-même; il s'agit du problème des arguments fonctionnels connu sous le nom "funarg problem" [Moo 70]. Le choix, dans Lisp, de la liaison dynamique, retenu uniquement pour des commodités d'implantation, entraine la perte de la propriété de transparence référentielle.

Une solution a été d'associer l'argument fonctionnel à son environnement en utilisant la fermeture(6). L'évaluation d'une fermeture (exp,env) conduit à l'évaluation de "exp" dans l'environnement "env".

4.2 La machine SECD [Lan 66]

La machine SECD est une architecture abstraite proposée en 1966 par P.Landin [Lan 66] basée sur la notion de β-réduction du lambda-calul. Elle décrit une sémantique opérationnelle et met en oeuvre l'appel par valeur et la liaison statique en représentant les arguments et résultats fonctionnels sous forme de fermeture.

---

(6) Terminologie inspirée du lambda calcul. Une expression sans variables libres est dite fermée.

Que se passe-t-il lorsqu'une variable libre apparaît dans une fonction? Elle peut être liée dans l'environnement de définition de cette fonction (langage à portée lexicale ou statique) ou dans un environnement d'exécution (langage à portée dynamique).

La plupart des interprètes Lisp actuels implantent une portée dynamique. En effet, ils utilisent une simplification qui consiste à dire que la valeur d'une abstraction (\x.N) est elle-même; il s'agit du problème des arguments fonctionnels connu sous le nom "funarg problem" [Mos 70]. Le choix, dans Lisp, de la liaison dynamique, retenu uniquement pour des commodités d'implantation, entraîne la perte de la propriété de transparence référentielle.

Une solution a été d'associer l'argument fonctionnel à son environnement en utilisant la fermeture(6). L'évaluation d'une fermeture (exp,env) conduit à l'évaluation de "exp" dans l'environnement "env".

4.2 La machine SECD [Lan 66]

La machine SECD est une architecture abstraite proposée en 1966 par P.Landin [Lan 66] basée sur la notion de β-réduction du lambda-calul. Elle décrit une sémantique opérationnelle et met en oeuvre l'appel par valeur et la liaison statique en représentant les arguments et résultats fonctionnels sous forme de fermeture.

---

(6) Terminologie inspirée du lambda calcul. Une expression sans variables libres est dite fermée

L'état de la machine est représenté par un quadruplet ‹S,E,C,D›, composé de:

- une pile (S) "Stack"
- un environnement (E) "Environment"
- Registre de contrôle (C) "Control"
- une poubelle (D) "Dump"

La pile sert à stocker les résultats intermédiaires.
L'environnement contient les liaisons de variables.
Le registre de contrôle sert à mémoriser le code à exécuter.
La poubelle est utilisée pour sauvegarder un état de la machine; il est donc lui-même de la forme ‹S',E',C',D'›.

Nous utilisons un exemple simple pour décrire le fonctionnement de ce type de machine.

les piles S, E, C, D sont disposées horizontalement, à chaque étape, avec le sommet à droite, les éléments séparés par des virgules, et les étapes sont disposées verticalement.

Soit à évaluer Exp = \x.\y. (+ x y) avec 3 et 4 comme arguments

	S	E	C	D
1.			Exp 3 4	
2.			ap, Exp 3, 4	
3.	4		ap, Exp 3	
4.	4		ap, ap, Exp, 3	
5.	4,3		ap, ap, Exp	
6.	4,3,E_1			
7.		(x,3)	\y. +xy	(4,(),ap,())
8.	E_2	(x,3)		(4,(),ap,())
9.	E_2,4		ap	
10.		(x,3),(y,4)	+xy	((),(),(),())
11.		(x,3),(y,4)	ap, +x,y	((),(),(),())
12.	4	(x,3),(y,4)	ap, +x	((),(),(),())
13.	4	(x,3),(y,4)	ap, ap, +, x	((),(),(),())
14.	4,3	(x,3),(y,4)	ap, ap, +	((),(),(),())
15.	4,3,+	(x,3),(y,4)	ap, ap	((),(),(),())
16.	4,+3	(x,3),(y,4)	ap	((),(),(),())
17.	7	(x,3),(y,4)		((),(),(),())
18.	7			

Dés que C et D sont vides ('7), l'évaluation de l'expression globale est terminée et le résultat se trouve en sommet de pile (18).

Nous avons utilisés les notations suivantes: E_1 et E_2 sont les fermetures. $E_1 \equiv$ (\y.+ x y, x, ())

$E_2 \equiv$ (+ x y, y, (x,3))

et 'ap' l'opérateur d'application

Nous avons présenté la version originale de la machine SECD. Celle-ci peut être grandement améliorée en compilant la gestion des variables. En effet, on peut connaître statiquement la place des variables dans l'environnement et ainsi éviter les parcours dynamiques de l'environnement. Une technique consiste à remplacer les noms de variables par un entier représentant leur position dans l'environnement. Cet entier représente le nombre de lambdas entre la variable et le lambda qui la lie et permet de remplacer les variables sans perdre d'information [DeB 72]. D'autres optimisations sont possibles, notamment pour éviter la construction de fermetures dans certains cas. Ces optimisations ainsi que le traitement de la récursivité ou l'introduction d'opérateurs primitifs sont présentés dans [Bur 75]. Ce modèle est efficace et il est devenu un standard pour la mise en oeuvre séquentielle des langages fonctionnels. Signalons notamment l'implantation de L.Cardelli [Car 83] qui s'inspire fortement de la machine SECD pour réaliser une implantation de ML qui constitue une référence en matière d'efficacité pour la mise en oeuvre de l'appel par valeur.

B- LES MACHINES A REDUCTION

Les mises en oeuvre utilisant les machines à réduction sont caractérisées par l'absence de séparation entre le code et l'environnement.

Elles sont basées sur la notion de réduction qui permet de remplacer une expression par une autre qui lui est mathématiquement (combinatoirement) équivalente. Le programme en langage applicatif est une expression qui peut être considérée

comme une description mathématique du résultat que souhaite obtenir l'utilisateur. La tâche de la machine à réduction est alors, en appliquant des règles qui préservent le sens de l'expression, de "simplifier" une expression jusqu'à l'obtention d'une forme "exploitable".

4.3 La SK machine [Tur 79a]

Turner utilise un algorithme d'abstraction pour transformer toute expression fonctionnelle en une expression équivalente dépourvue de variables et composée des constantes du programme original et de combinateurs [Tur 79b].
Ainsi la notion d'environnement avec les liaisons variable-valeur disparait et le processus d'abstraction peut être vu comme la compilation de l'accès aux variables.

4.3.1 Réduction par graphe

Les expressions combinatoires se représentent aisément sous la forme d'un graphe, les noeuds indiquant l'application d'une fonction à son argument et les feuilles les atomes.
Nous présentons, à titre d'exemple, le graphe correspondant à la fonction Som.

Som(n) = (IF ((= n 1) 1) (True (+ n (Som (- n 1)))))

On obtient après abstraction le code combinateur

Som = S (C (B IF (= 1)) 1) (S+ (B som (C -1)))

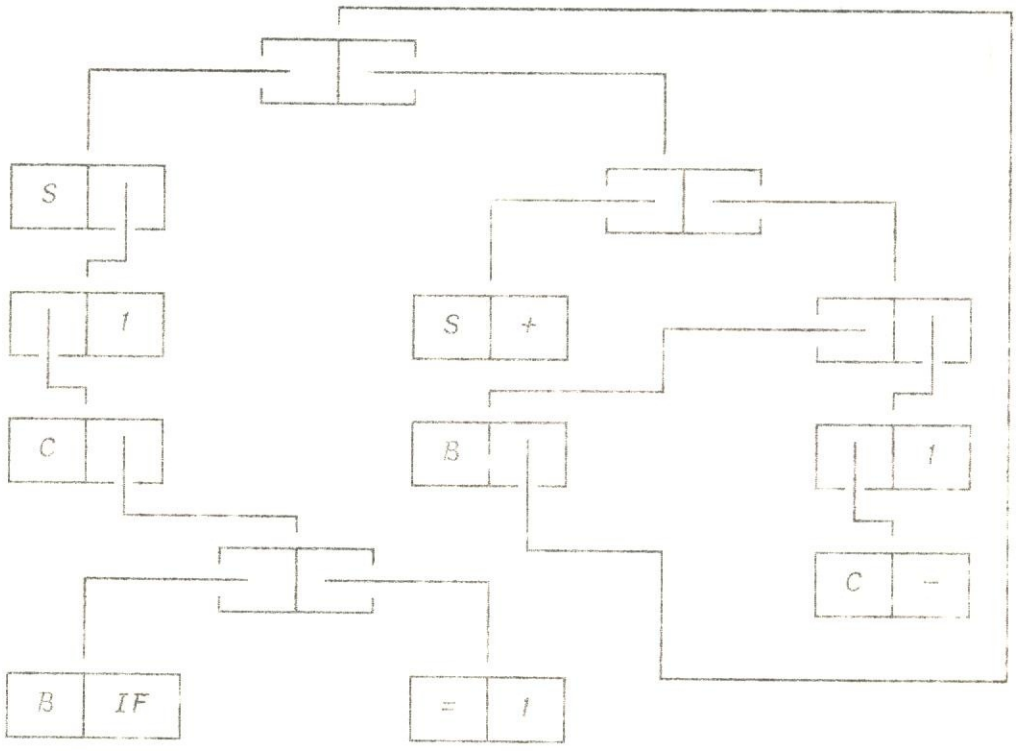


Fig 1

La phase d'exécution réalisée par la SK machine consiste à transformer progressivement cette structure en appliquant les règles de réduction jusqu'à l'obtention d'un objet non réductible:

S f g x	=> f x (g x)	K x y	=> x
C f g x	=> (f x) g	B f g x	=> f (g x)
I x	=> x	U f (P x y)	=> f x y
IF True x y	=> x	IF False x y	=> y
PLUS m n	=> m+n		

On définit de façon similaire MULT, MOINS, DIVISE, etc.

Ces règles de réduction peuvent être implantées comme des règles de transformation de graphe: le noeud correspondant au membre gauche d'une règle de réduction est remplacé par le membre droit. la figure 2 traite le cas de la règle de réduction associée à S.

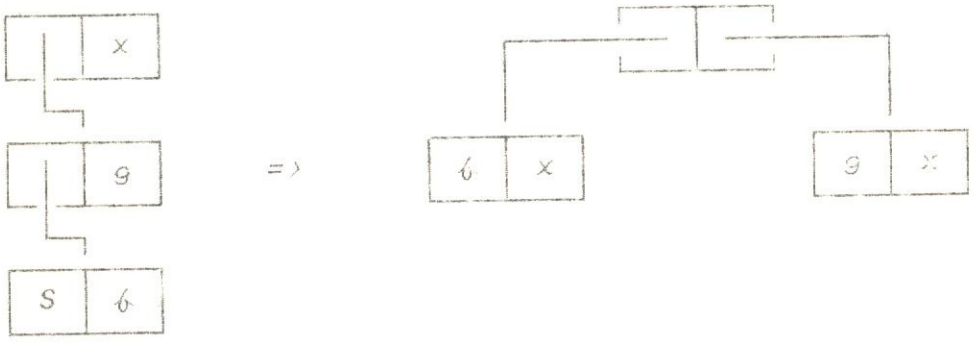
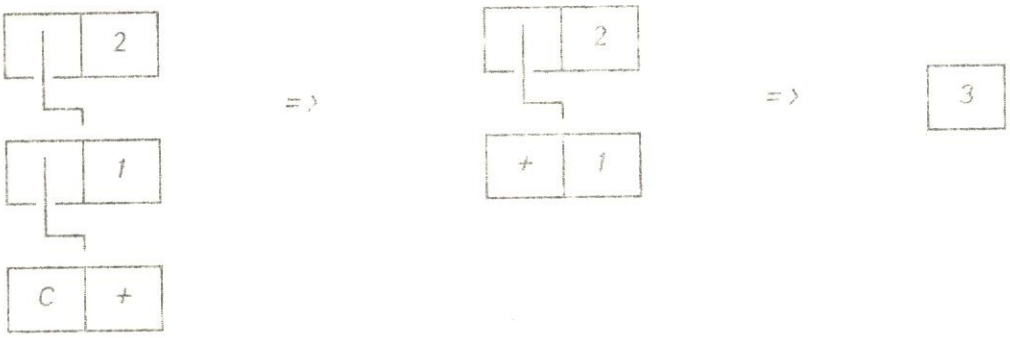


fig 2

Exemple: Calcul du successeur de 2, on obtient grâce a l'algorithme d'abstraction le code: C (+ 1) 2

A partir de la structure de graphe correspondant au code produit par le compilateur, on obtient la réduction de graphe suivante:



4.3.2 L'ordre de réduction

Deux types de stratégies de réduction seront présentés:

- l'ordre normal: consistant à réduire le redex le plus à gauche et le plus externe correspondant à l'appel par nom.

- l'ordre applicatif: consistant à réduire le redex le plus à droite et le plus interne correspondant à l'appel par valeur.

D'un point vue théorique nous savons d'une part que le résultat d'une chaîne de réduction est indépendant de la stratégie choisie. D'autre part si une terminaison existe, l'appel par nom permet systématiquement de l'obtenir ce qui n'est pas forcément le cas de l'appel par valeur. En effet l'évaluation d'une expression (K 2 X) où X est une expression dont l'évaluation n'aboutit pas, l'appel par nom rendra le résultat 2 en une étape de réduction alors que l'appel par valeur tentera en vain de réduire l'expression X.

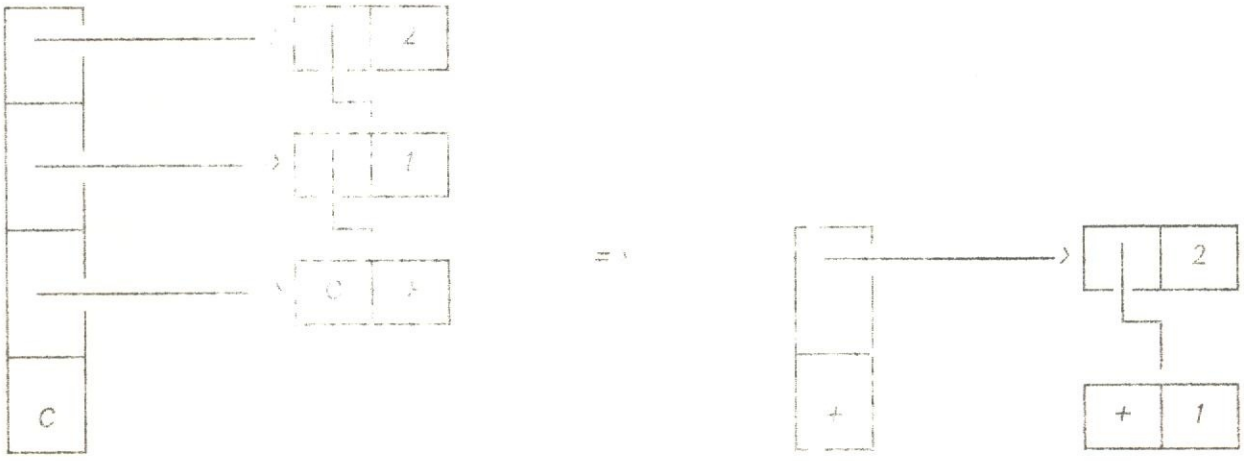
Cependant, si l'appel par nom est sûr, il est coûteux dans le cas où un argument de fonction est en général substitué sous forme non évaluée dans le corps de cette fonction, ce qui peut entraîner une réévaluation de cet argument autant de fois qu'on le rencontrera par la suite.

Le choix d'une représentation des expressions combinatoires sous forme de graphes, dans la machine à réduction, permet de partager une sous-expression en utilisant des pointeurs. Ainsi, après une substitution, toutes les occurrences de l'argument dans le corps de la fonction sont des pointeurs sur une même

réduire les arguments de Plus à des nombres et pouvoir appliquer la règle de réduction appropriée.

Exemple: Voici le comportement de la pile d'exécution durant la première étape de réduction de l'exemple précédent. La pile est représentée avec le sommet de pile en bas dans notre schéma.

Avant la réduction, nous avons la configuration suivante:



Quand l'objet rencontré en tête de pile est un objet imprimable, on s'arrête. Si l'utilisateur soumet pour l'évaluation une expression dont le résultat est une structure de données, par exemple une liste d'objets, l'algorithme de réduction laissera en tête de pile une représentation de cette structure dont toutes les composantes n'ont pas forcément toutes été réduites. La procédure d'impression se charge alors d'appeler l'algorithme de réduction pour simplifier chacune des composantes afin de pouvoir imprimer cette structure. Si l'une des composantes est elle même une structure de données, ce processus se répète récursivement jusqu'à l'obtention de l'impression complète.

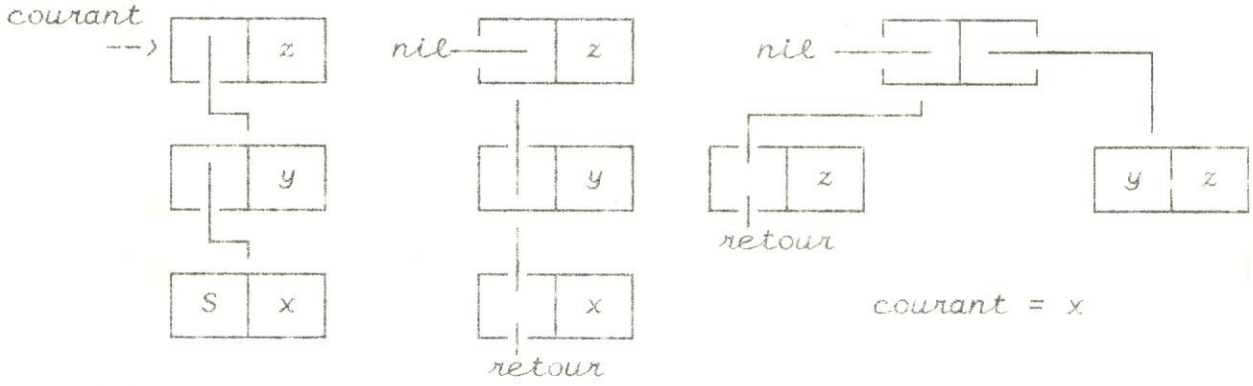
4.4 La Machine SKIM [Cla 80] et [Sto 84]

La SKIM (S,K,I reduction Machine) [Cla 80][Sto 84] consiste en une traduction au niveau matériel des idées de Turner. L'objectif initial du projet était de montrer qu'avec très peu de matériel spécialisé, il est possible d'améliorer les performances.

L'architecture de cette machine est simple et relativement classique. Il s'agit d'une machine micro-programmée dont les micro-instructions sont spécialisées dans la manipulation d'arbres. En particulier, les micros-instructions peuvent réaliser en parallèle des tests et des branchements, et des transferts de données.

Les combinateurs constituent le code machine et ils sont traités par le micro-code. Ce micro-code inclut la gestion de mémoire, celle-ci étant organisée en paires. Cette mémoire est bien adaptée à la représentation des arbres de combinateurs nécessitant uniquement des cellules binaires. Cette mémoire est divisée en deux bancs: "head" et "tail", une adresse étant composée du numéro du banc et d'une adresse dans celui-ci.

La réduction d'un arbre de combinateurs est réalisée en parcourant les fils gauches de l'arbre jusqu'à un combinateur. Il est nécessaire de mémoriser les noeuds ainsi visités afin de retrouver les arguments du combinateur à réduire ainsi que le noeud à ré-écrire. La technique mise en oeuvre dans la SKIM ne demande pas l'utilisation d'une pile pour cette mémorisation. De plus l'accès aux arguments reste très simple grâce au parcours de la liste inversée.



pointeurs renversés et exécution

Cette idée a également été appliquée pour l'évaluation des arguments d'une fonction stricte (les arguments sont évalués avant l'appel), ce qui est le cas de la plupart des fonctions prédéfinies.

Le calcul des arguments d'une fonction stricte nécessite un appel récursif de l'évaluateur, ce qui implique bien évidemment une sauvegarde de son état complet. Le principe consiste à évaluer les arguments en laissant dans la chaîne de retour une marque qui représente un pseudo combinateur associé à un point d'entrée dans le micro-code. Les arguments sont ainsi évalués récursivement avec un minimum de sauvegarde d'état de l'évaluateur et aucune structure de donnée supplémentaire n'est nécessaire ; tout se passe sur le graphe lui-même. Le gain apporté a été de 10% [Lag 87].

4.5 Machine à réduction par graphe pour VAAL [Bou 90](7)

Nous proposons une machine à réduction par graphe opérant sur un code combinatoire T_{CE}, obtenu en sortie de l'algorithme d'abstraction de VAAL. Ce code est formé par une imbrication d'expressions parenthèsées de la forme $(M_1 \ldots M_n)$ dénotant l'application de M_1 aux arguments $M_2 \ldots M_n$.
L'expression est dite élémentaire si tous les M_i sont des entités atomiques.

On appellera expression composée une expression parenthèsée dont les éléments (application ou argument) peuvent eux-même être des expressions parenthèsées.

4.5.1 Représentation des expressions combinatoires.

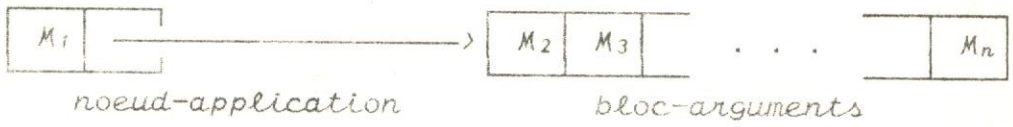
Etant donnée une forme élémentaire donnée $(M_1\ M_2 \ldots M_n)$, nous proposons une représentation graphique reflétant, dans sa sémantique:

- que le premier élément de l'expression parenthèsée joue le rôle de l'application: il sera mis dans un noeud application.
- que les éléments suivants sont les arguments du premier élément.

---

(7) Cette machine a été développée au sein de l'équipe SEEP dans le cadre du projet VAAL.

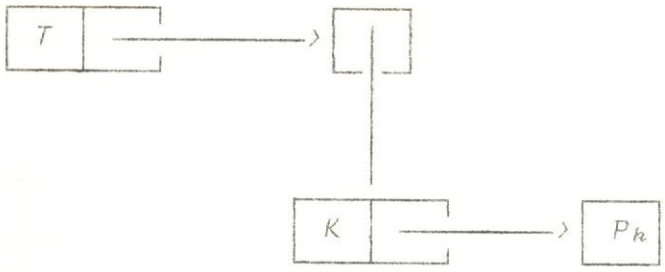
Ils sont introduits dans un vecteur $M_2 \ldots M_n$.



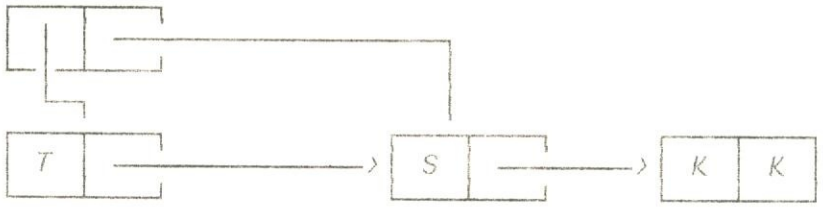
Exemples:

i) expression élémentaire: $P_1 \equiv (S\ K\ K)$

ii) expression composée: $P_{h+1} \equiv (T\ (K\ P_h))$



iii) expression composée $((T\ (S\ K\ K))\ (S\ K\ K))$



4.5.2 Le processus de réduction:

L'ordre de réduction choisi pour cette machine à réduction tient de l'ordre normal (les arguments ne sont évalués que nécessairement) et de l'ordre applicatif (les arguments sont évalués au plus une fois). Il nécessite donc le passage de la représentation par arbre à une représentation par graphe grâce à laquelle plusieurs noeuds-application permettent éventuellement de se chainer à un même bloc-arguments, évitant ainsi la duplication coûteuse de blocs identiques.

Le processus de réduction d'un graphe consiste alors à appliquer à chaque pas la règle de réduction associée à la feuille la plus à gauche, un pas de réduction consistant en la ré-écriture d'un sous-arbre.

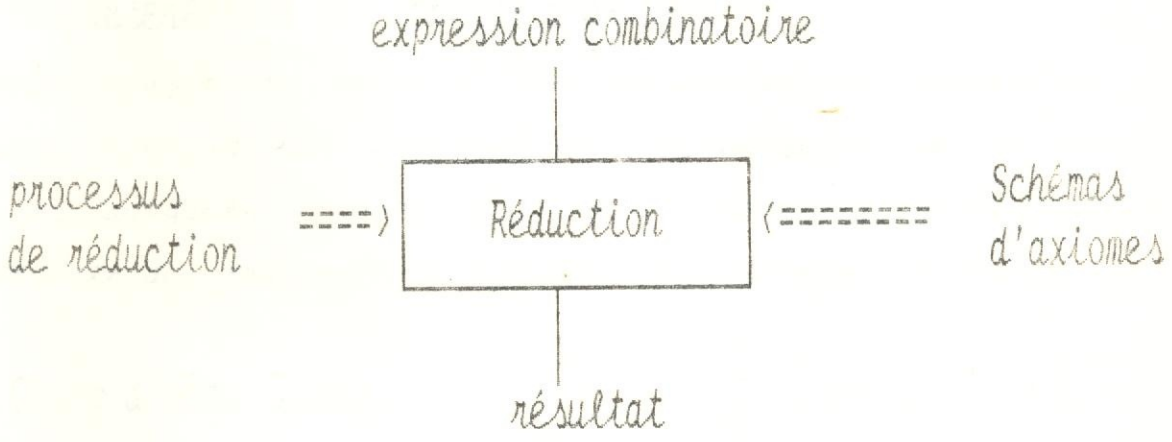
Exemple: Réduire ((S K K) T T)

On obtient ((S K K) T T) => T

Les phénomènes de "fuite de mémoire" (quand la mémoire disponible disparait de façon invisible) ou de "freinage" (quand l'exécution d'une fonction est ralentie) entraînent un surcoût d'espace. Cela implique des appels répétés au glaneur de cellules. Un surcoût en temps est alors associé à cet usage excessif de mémoire. Il à noter que l'introduction de notions telles les chaines de directions [Ken 82] [Sle 82] permettent d'avoir des graphes plus compacts au prix d'une plus grande complexité de la machine.

Chapitre V: Machine à réduction matricielle

La machine à réduction constitue le support d'implantation du langage. Pour cela, elle doit permettre la définition et la représentation de tous les éléments qui contribuent à la réalisation de la réduction. Ainsi que nous l'avons vu lors de la présentation de la machine à réduction par graphe, il apparait nécessaire de disposer, du schéma de la machine à réduction:



- d'une représentation des schémas d'axiomes,
- d'une représentation en mémoire de l'expression combinatoire
- d'une définition du processus de réduction.

Dans le cas de la machine à réduction par graphe étant donnée une expression combinatoire, représentée sous forme de graphe, le processus de réduction élémentaire est la substitution d'un sous graphe par son graphe "simplifié".

Nous proposons, maintenant, une nouvelle approche de machine à réduction. Le processus d'implantation devient un processus algébrique dans le sens où la représentation utilisée est la matrice et le processus de réduction est le produit matriciel. La machine à réduction matricielle repose alors sur:

- la représentation, sous forme de matrices, des schémas d'axiomes, et
- l'utilisation du produit matriciel pour réaliser le processus de réduction.

A- Représentation des schémas d'axiomes:

Il s'agit de définir une représentation permettant de prendre en charge des combinateurs non currybiées de la théorie TGE et auxquels correspondent des schémas d'axiomes faisant intervenir des séquences de longueurs non fixées à l'avance.

5.1 Champ d'occurrences:

Dans ce qui suit un champ d'occurrences de longueur n représentera une suite de places vides numérotées 1, 2, ..., n.

Nous associons à chaque schéma d'axiome un champ d'occurrences qui permet de le représenter comme un "processus d'affectation d'arguments" dans un champ d'occurrences.

Nous étudions, à titre d'exemple, les processus d'affectation d'occurrences et d'arguments aux occurrences associés au schéma d'axiome du combinateur T.

$((T\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow (\ (\ G_1\ X_1 \ldots X_n\ )\ \ X_2 \ldots X_n\ )$

champ d'occurrences: 1 2 3 4 5 6 7

- le champ d'occurrence est de longueur 7.
- La parenthèse '(' est affectée aux occurrences 1 et 2.
- La parenthèse ')'est affectée aux occurrences 5 et 7.
- Le terme G_1 est affecté à l'occurrence 3.
- La séquence $X_1 \ldots X_n$ est affectée à l'occurrence 4 et
- La séquence $X_2 \ldots X_n$ est affectée à l'occurrence 6.

5.2 Opérations sur les arguments

Chaque schéma d'axiome associé à un combinateur M est de la forme:

$((M\ X_1 \ldots X_n)\ Y_1 \ldots Y_m) \Rightarrow \alpha(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$

où $\alpha(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ intègre des sous-séquences de $X_1 \ldots X_n$ et $Y_1 \ldots Y_m$, et/ou des termes X_i ou Y_j.

Nous utiliserons les mêmes conventions $[(-/+)i]$ et $[(+/-)i,(+/-)j]$ que dans VAAL pour représenter la sélection d'argument et l'extraction de sous-séquences.

Exemples: Etant donnée une séquence $X_1 \ldots X_n$, on peut:

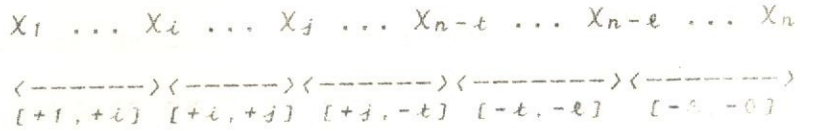
- Sélectionner un argument

  $[-i]$: sélectionne le ième argument à partir du dernier: X_{n-i}

  $[+i]$: sélectionne le ième argument: X_i

  $[-0]$: sélectionne le dernier argument: X_n

- Extraire une sous-séquence



Dans l'exemple de T en introduisant les conventions définies ci-dessus, le processus d'affectation de champ d'occurrences devient:

- '(' est affectée aux occurrences 1 et 2.
- ')'est affectée aux occurrences 5 et 7.
- [1] est affectée à l'occurrence 3 (relative à la première séquence d'arguments de T)
- [+1,-0] est affectée à l'occurrence 4 (relative à la deuxième séquence d'arguments de T) et
- [+2,-0] est affectée à l'occurrence 6 (relative à la deuxième séquence d'arguments de T).

On définit de manière identique le champ d'occurrences associé à chaque atome (S, K, T, L, et D).

On associe deux champs d'occurrences pour le combinateur D selon que la taille de la deuxième séquence soit égale à 1 ou supérieur à 1.

La taille du champ d'occurrences pour le combinateur S, dépendra de la taille de la première séquence.

De plus, si le combinateur apparait dans le contractum alors il lui est réservé une place dans le champ occurrences.

5.3 Représentation matricielle

Le type de produit matriciel que nous allons définir dans la partie B, nous amène à représenter un combinateur E, auquel est associé un schéma d'axiome, comme une matrice M_E à p lignes et q colonnes où

- p est le nombre de séquences d'arguments appliqué à E augmenté de deux.
- q la taille du champ d'occurrences.
- La première ligne de M_E indique les occurrences des parenthèses.

0 à la $j^{ème}$ colonne si aucune parenthèse n'est affectée à la $j^{ème}$ occurrence.

1 (resp. -1) à la $j^{ème}$ colonne si "(" (resp. ")") est affectée à la $j^{ème}$ occurrence.

- La $i+1^{ème}$ ligne ($1 \leq i \leq p-1$) est associée à la $i^{ème}$ séquence d'arguments appliquée au combinateur E.

0 à la $j^{ème}$ colonne si aucune sous-séquence correspondant à la $i^{ème}$ séquence n'est affectée à la $j^{ème}$ occurrence du champ de E.

[(+/-)i,(+/-)j] ou [(+/-)i] à la $j^{ème}$ colonne si la sous-séquence correspondant à la $i^{ème}$ séquence est affectée à la $j^{ème}$ occurrence du champ de E.

- La dernière ligne correspond à la récursivité

1 à la $j^{ème}$ colonne si le combinateur E apparait à la $j^{ème}$ occurrence du champ de E.

En procédant de la même manière que précédemment, on associe aux combinateurs K, T et L les matrices:

$((K\ X_1 \ldots X_n)\ Y_1 \ldots Y_m) \Rightarrow X_1$

$$M_K = \begin{bmatrix} 0 \\ [1] \\ 0 \\ 0 \end{bmatrix}$$

$((T\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow ((G_1\ X_1 \ldots X_n)\ X_2 \ldots X_n)$

$$M_T = \begin{bmatrix} 1 & 1 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & [+1] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & [+1,-0] & 0 & [+2,-0] & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$((L\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow ((G_1\ X_1 \ldots X_n)\ (G_2\ X_1 \ldots X_n)\ X_1 \ldots X_n)$

$$M_L = \begin{bmatrix} 1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & [+1] & 0 & 0 & 0 & [+2] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & [+1,-0] & 0 & 0 & 0 & [+1,-0] & 0 & [+1,-0] & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Dans le cas du combinateur D, auquel correspond deux schémas d'axiomes, on associe deux matrices:

$((D\ G_1 \ldots G_m)\ X) \Rightarrow (G_1\ X)$

$((D\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow (G_2\ X_1\ X_2 \ldots X_n)$

$$M_{D1} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & [+1] & 0 & 0 \\ 0 & 0 & [+1,-0] & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{ et } M_{D2} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & [+2] & 0 & 0 \\ 0 & 0 & [+1,-0] & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Quant au combinateur S, dont le schéma d'axiome fait intervenir au niveau du contractum une séquence de séquences de termes, on associe la famille de matrices:

$((S\ G_1 \ldots G_m)\ X_1 \ldots X_n) \Rightarrow ((G_1\ X_1 \ldots X_n)\ (G_2\ X_1 \ldots X_n) \ldots (G_m\ X_1 \ldots X_n))$

$$M_{Sm} = \begin{bmatrix} 1 & 1 & 0 & 0 & -1 & \cdots & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & [+1] & 0 & 0 & \cdots & 0 & [+m] & 0 & 0 & 0 \\ 0 & 0 & 0 & [+1,-0] & 0 & \cdots & 0 & 0 & [+1,-0] & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \left(\sum_{i=1}^{m} \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & [+i] & 0 & 0 \\ 0 & 0 & [+1,-0] & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right) \Rightarrow = \left(\sum_{i=1}^{m} S_i \right)$$

où Σ représente la concaténation horizontale des matrices S_i et en identifiant:

"(" avec $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ et ")" avec $\begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

B- Processus de réduction:

L'expression combinatoire que l'on obtient comme résultat de l'algorithme d'abstraction sera représentée sous forme de chaîne. Ce choix s'accorde bien avec l'utilisation du produit matriciel: la chaîne peut être alors traitée comme un vecteur.

5.4 Stratégie de réduction:

Dans la conception proposée, on peut choisir entre différentes stratégies de réduction, dont l'ordre normal ou l'ordre applicatif. En effet, le théorème de Church Rosser assure l'indépendance par rapport au schéma d'execution retenu.

Dans notre machine à réduction, la réduction consiste en un produit matriciel, faisant intervenir lors de son calcul les opérations + et * portant sur des suites finies de symboles qu'on appellera expression.

5.5 L'algèbre des opérations

Nous définissons l'algèbre des opérations + (concaténation) et * (produit) comme suit:

X+[] = []+X = X X une expression et [] partie vide

$X+Y = XY$ X et Y des expressions

$X*0 = []$ X une expression et $[]$ partie vide

$X*1 = X$ X une expression

$(*-1 =)$

$X_1,\ldots,X_n*[+i,+j] = X_i,\ldots,X_j$ $(1\leq i\leq n,\ 1\leq j\leq n)$

$X_1,\ldots,X_n*[+i,-j] = X_i,\ldots,X_{n-j}$ $(i\geq 1,\ j\geq 0,\ i+j\leq n)$

$X_1,\ldots,X_n*[-i,+j] = X_{n-i},\ldots,X_j$ $(i\geq 0,\ j\geq 1,\ i+j\leq n)$

$X_1,\ldots,X_n*[-i,-j] = X_{n-i},\ldots,X_{n-j}$ $(0\leq i\leq n,\ 0\leq j\leq n)$

$X_1,\ldots,X_n*[+i] = X_i$ $(1\leq i\leq n)$

$X_1,\ldots,X_n*[-i] = X_{n-i}$ $(0\leq i<n)$

5.6 Produit matriciel

Etant donné un combinateur E parmi S, K, T, L, D auquel on applique les séquences $X_1\ldots X_n$ et $Y_1\ldots Y_m$, la réduction du redex $((E\ X_1\ldots X_n)Y_1\ldots Y_m)$ s'obtient à partir du produit matriciel, noté V . M_E, entre le quatre-uplet $V \equiv [(\ X_1,\ldots,X_n\ \ Y_1,\ldots,Y_m\ E]$ et la matrice M_E à quatre lignes définie comme suit:

M_K si $E \equiv K$

M_T si $E \equiv T$

M_L si $E \equiv L$

M_{D1} si $E \equiv D$ et $m = 1$

M_{D2} si $E \equiv D$ et $m > 1$

M_{Sm} si $E \equiv S$

Le résultat du produit matriciel est alors un vecteur $L \equiv [L_1\ L_2\ \ldots\ L_\ell]$ correspondant au contractum $L_1L_2\ldots L_\ell$.

Exemple: Réduire ((T 1 2 3) 2 5)

$$V \,.\, M_T = [(\ 1,2,3\ 2,3\ T] \,.\, \begin{bmatrix} 1 & 1 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & [+1] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & [+1,-0] & 0 & [+2,-0] & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= [(*1\ (*1\ 1,2,3*[+1]\ 2,3*[+1,-0]\ (*-1\ 2,3*[+2,-0]\ (*-1\ ]$$

$$= [(\ (\ 1\ 2,3\ )\ 3\ )]$$

Donc ((T 1 2 3) 2 5) => ((1 2 3) 3)

On remarque que le produit matriciel V.Ms se ramène à une concaténation horizontale des vecteurs résultats des produits matriciels V.si, i=1,...,m. Plus précisément V.Ms = (Σ V Si).

Exemple: Réduire ((S K K) 1 2)

$$V.M_S = (\sum_1^2 V.S_i) = (\ [(\ K,K\ 1,2\ S] \,.\, \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & [+1] & 0 & 0 \\ 0 & 0 & [+1,-0] & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$+\ [(\ K,K\ 1,2\ S] \,.\, \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & [+2] & 0 & 0 \\ 0 & 0 & [+1,-0] & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\ )$$

$$= ([(*1\ K, K*[+1]\ 1,2*[+1,-0]\ (*-1]$$
$$+ [(*1\ K, K*[+2]\ 1,2*[+1,-0]\ (*-1\ ])$$
$$= ([(\ K\ 1,2\ )\ (\ K\ 1,2\ )]\ )$$

Donc ((S K K) 1 2) => ((K 1 2) (K 1 2)), et la réduction se poursuit comme suit:

$$[(\ 1,2\ (K\ 1\ 2)\ K] \cdot \begin{bmatrix} 0 \\ [+1] \\ 0 \\ 0 \end{bmatrix} = [\ 1,2*[+1]\ ] = [\ 1\ ]$$

par conséquent ((K 1 2) (K 1 2)) => 1

et Finalement ((S K K) 1 2) => 1

Remarque: De la même manière, aux combinateurs définis dans l'algorithme d'abstraction de VAAL, on peut faire aussi correspondre une représentation matricielle pour permettre au processus de réduction d'être ramené à un produit matriciel.

5.7 Un cycle de réduction

Etant donné un ensemble récursif C de combinateurs admettant une représentation matricielle, la réduction d'une expression combinatoire est réalisée selon le cycle suivant:

1. Sélection d'un atome E ε C (par exemple le plus à gauche et le plus externe: stratégie d'ordre normal), s'il existe (sinon la réduction est achevée).
2. Construction du vecteur V correspondant aux arguments de E.
3. Calcul du produit matriciel V . ME.

4. Transformation du vecteur résultat L en une expression combinatoire.

5. Aller en 1

Le cycle est donc achevé dés l'obtention d'une expression irréductible.

Basée sur des opérations algébriques (arithmétiques), la machine à réduction matricielle, est mieux adaptée à la machine Von-Neumann.

Ses caractéristiques sont prometteuses pour le langage VAAL:

- Toutes les matrices introduites sont creuses: au niveau d'une colonne, il n'y a qu'un seul élément différent de zéro. Ce qui permet une représentation plus compacte utilisant un espace réduit. La taille des matrices étant réduite, le nombre d'opérations du produit matriciel sera minimal. Ce qui limite l'accés à la mémoire et le temps de réponse.
- L'expression combinatoire sous la forme de chaîne permet une représentation vectorielle qui consomme peu d'espace mémoire.

Une comparaison sommaire avec la machine à réduction par graphe montre les avantages évidents de la machine à réduction matricielle au niveau de la consommation mémoire, et laisse entrevoir des performances bien supérieures. Cette affirmation devrait être confirmée par une comparaison de performance systématique sur un matériel adéquat.

CONCLUSION

Les modèles mathématiques associés aux langages fonctionnels, tels que les théories combinatoires, basées sur la notion primitive de fonction en tant que processus de calcul, rejetant la définition ensembliste qui est restrictive, a donné un nouvel élan à la programmation fonctionnelle. Malheureusement, la currylication imposée par ces modèles, si elle permet de prendre en charge les fonctions polyadiques, pose des problèmes pour les fonctions à arité variable, alors que les langages fonctionnels le font au prix d'une simple structuration.
La notion d'arité variable, bien qu'intuitivement simple, s'avère délicate à définir. Une meilleure compréhension, sinon maîtrise, de cette notion, permet dans la mesure où l'on peut développer un algorithme d'abstraction correspondant, d'enrichir la classe des fonctions utilisateurs.

Le principal reproche habituellement fait aux langages fonctionnels se situe au niveau des performances, tributaires de la machine de Von-Neumann qui n'est pas adéquate au modèle fonctionnel. En attendant de disposer de machines fonctionnelles du type de la machine MaRS [MaR 86], nous avons conçu une machine à réduction matricielle. Cette machine est mieux adaptée au modèle Von-Neumann dans la mesure où le processus de réduction devient un produit matriciel.

Conclusion

Ce travail constitue un support au développement d'un environnement visant l'intégration dans VAAL de concepts logiques et orientés objets. De plus, nous nous intéressons aux perspectives de développement suivantes:

- Conception d'un compilateur du langage VAAL utilisant la technique d'évaluation partielle [Con 89].

- Une implantation parallèle de la machine à réduction matricielle.

BIBLIOGRAPHIE

[All 78] J. Allen
Anatomy of Lisp
McGraw-Hill, New-York

[Abd 76] S.K. Abdali
An abstraction algorithm for combinatory logic
Journal of symbolic logic, vol.41, n°1, mars 76

[Bac 78] J.W. Backus
an programming be liberated from the Von Neumann style?
A functional style and its algebra of program.
Communications of the ACM Vol.21, n° 8, 1978, pp613-641

[Bel 86] P. Bellot
Sur les sentiers du GRAAL. étude, conception et réalisation d'un langage de programmation sans variable.
Thèse de 3ème cycle. Univ. Pierre et Marie Curie. Paris.

[Bel 87] P. Bellot, V. Jau
A theory of natural modelisation and implementation of functions with variable arity.
Int conf on functional programming languages and computer architecture (FPLCA) Portland, sept 87. LNCS 274 pp212-233.

[Bel 88] K. Belhir
Conception d'une machine orientée fonctions, application a l'implantation d'un langage dirigé par les données.
Thèse de docteur Paris VI

[Bel 89] P. Bellot, D. Sarni
Proposal for a natural formalization of functional programming concepts.
RAIRO, theoretical informatics and applications, vol.22, n° 3, pp343-360, 1988.

[Ben 89] C.B. Ben-Yelles, K. Benabadji
VAAL
Rapport interne, Dépt. d'inf. fondamentale, USTHB

[Ben 90] C.B. Ben-Yelles, C. Bouabana
Types principaux dans TGE
ICM 90, Kyoto, Japon

[Bou 90] Y. Boumahdi, A. Mahiout
Conception et réalisation d'une machine à réduction pour le langage VAAL
Mémoire d'ingenieur, Institut d'informatique (USTHB).

Bibliographie

[Bou 90a] C.Bouabana
Stratification dans T_{GE}
Thèse de Magister. Institut d'infor. (D.I.F.), USTHB

[Bur 75] W.H.Burge
Recursive programming techniques
Addison-Wesley. 1975

[Car 83] L.Cardelli
The functional abstract machine.
Polymorphism. Vol.1. n° 1

[Cha 84] J.Chailloux
Le-Lisp de l'INRIA:le Manuel de référence.
INRIA. Le chesnay

[Chu 41] A.Church
The calculi of lambda conversion
Princeton University Press

[Cla 80] T.J.W Clarke. P.J.S.Gladstone, C.D.MacLean. A.C.Norman
SKIM - The S, K, I. Reduction Machine
Proc. first Int ACM Lisp Conf. Stanford. 1980. pp 128-135

[Con 89] C.Consel
Analyse de programmes, évaluation partielle et génération de compilateurs.
thèse de docteur. Paris VI.

[Cur 33] H.B.Curry
Apparent variables from the standpoint of combinatory logic.
Annals of mathematics (2), Vol 34. 1933, pp 381-404

[DeB 72] N.G.De Bruijn
Lambda-calculus notation with nameless dummies. a tool for automatic formula manipulation
Indag Math. 34, 1972, pp 381-392

[Dur 86] M-H.Durand
Etude et evaluation du parallélisme dans les langages fonctionnels. Une approche de la réduction de graphe par les combinateurs.
thèse de Docteur ingénieur. ENSAE, Toulouse

[Fai 82] J.Fairbairn
Ponder and its type system.
Technical report 31. Computer Lab., Cambridge

[Fra 88] P.FRADET
Compilation des langages fonctionnels par transformation de programmes
These de docteur en Informatique de l'univ. de Rennes I.

[Gla 84] H.Glaser. C.Hankin. D.Till
Principales of functional programming
Prentice-Hall Inter.

[Hin 86] J.R.Hindley. J.P.Seldin
Principal type-schemes for functional programs
Proc 9th ACM sym. on principles of programming langages.

[Hua 82] R.J.M.Hughes
Super-combinators: A new implementation method for applicative languages.
Conf record of the 1982 ACM symposium on Lisp and Functional programming, Pittsburg Aout 82.

[Hug 84] R.J.M.Hughes
The design and implementation of programming languages
PhD thesis. PRG 40. Programming research group. Oxford

[Jay 89] V.JAY
Modélisation des opérateurs d'arité variable en logique Combinatoire. Application au formalisme fonctionnel.
Thèse de docteur en informatique de l'Univ. Paris 6.LITP

[Joh 84] T.Johnsson
Efficient compilation of lazy evaluation.
In proc of the ACM conf on compiler construction. Montreal, pp 58-69.

[Ken 82] J.R.Kennaway. M.R.Sleep
Director strings as combinators
Dept. of computer science. Univ of East Anglia

[Lag 87] F.Lagnier
Experience de mise en oeuvre et d'utilisation d'un langage fonctionnel
These de docteur en informatique de l'institut national polytechnique de Grenoble. Octobre 1987

[Lan 66] P.J.Landin
The next 700 programming Languages
Communications of the ACM. Vol.9. pp.157-166. Mars 1966

[MaD 86] M.Lemaitre. M.Castan. M.-H.Durand. G.Durrieu. B.Lecussan
Mechanisms for efficient multiprocessor combinator reduction
Conference record of the 1986 ACM symposium on Lisp and functional programming. Cambridge. Massachusetts.

[MaD 87] M.Lemaitre. M.Castan. M.-H.Durand
A set of combinators for abstraction in linear space.
Information Proc Letters, vol.24. pp183-188. North-Holland.

[McC 62] McCarty
The Lisp 1.5 programmers manual
MIT press. Cambridge. Mai 1962