Exécution en λ-calcul avec ressources

Étienne MIQUEY

Juin/Juillet 2010

Stage effectué sous la direction d'Emmanuel BEFFARA et Lionel VAUX à l'Institut Mathématique de Luminy.

L'objectif de ce stage était d'adapter la machine de Krivine existant pour le λ-calcul usuel afin d'en faire une machine abstraite pour le λ-calcul avec ressources, puis de l'implémenter.

Dans ce rapport, on commencera dans un premier temps par se familiariser avec le λ-calcul avec ressources, avant de s'intéresser à la machine abstraite conçue pendant ce stage. On se concentrera ensuite sur le problème de sa correction ; puis, dans un dernier temps, on s'intéressera à son raffinement, que ce soit d'un point de vue pratique, afin de rendre l'exécution la plus efficace possible, ou de façon plus théorique, dans le but de la relier notamment à des notions de normalisation.

Remerciements

Je tiens à remercier Emmanuel et Lionel, tant pour le sujet de stage fort intéressant qu'ils m'ont proposé que pour leur disponibilité durant ledit stage. Je remercie plus particulièrement Lionel pour les envolées au tableau dans le monde de la sémantique dénotationnelle et Emmanuel pour ses conseils avisés sur mes prototypes successifs de machine abstraite. Ils m'ont également permis d'assister à la semaine "Réalisabilité à Chambéry et à la Choco-party de Juin, qui furent l'occasion de découvrir le vrai visage de bon nombre de logiciens d'ici et là.

Merci également à toute l'équipe Logique de Programmation à l'IML pour son accueil, et l'intérêt d'un certain nombre porté à mon humble stage. Plus spécialement, je remercie Thomas et Pierre qui m'ont généreusement laisser écrire tout un tas de bêtises au tableau jusqu'à sa saturation.

Enfin, il est de rigueur que je salue ici la beauté des calanques de Sugiton et Morgiou, agréments appréciables et appréciés du séjour, sans lesquelles ce n'aurait pas vraiment été pareil...

Table des matières

1 Introduction : Le λ-calcul avec ressources **4**
1.1 Idée & origine 4
1.2 Syntaxe 4
1.3 Réduction 5

2 Une machine abstraite pour le calcul avec ressources **6**
2.1 La machine de Krivine usuelle 6
2.2 Idées générales 7
2.3 Ladite machine 7

3 Correction de la machine **9**
3.1 Schéma de principe 9
3.2 Fonctions de traduction/décompilation 10
3.3 La preuve 12
3.4 Normalisation 16

4 Discussion **16**
4.1 Une normalisation plus forte 16
4.2 Lien avec la solvabilité 17
4.3 Implémentation, optimisation 18
4.4 Une autre version de la machine 19

5 Conclusion **20**

1 Introduction : Le λ-calcul avec ressources

1.1 Idée & origine

Le λ-calcul avec ressources est une forme dérivée du λ-calcul avec multiplicités [Bou93], dont l'idée était de compter la multiplicité des arguments. En effet, lors de la β-réduction usuelle $(\lambda x.M)N \to M\{N/x\}$, l'argument N est dupliqué autant de fois que nécessaire. En raffinant ainsi le λ-calcul, certaines équivalences observationnelles ne tiennent plus. Ainsi, si dans le λ-calcul standard, on a $xx = x(\lambda y.xy)$, ceci devient faux en rajoutant les multiplicités $(xx^\infty \neq x(\lambda y.xy^\infty)^\infty)$: dans le contexte où $\{(\lambda z.z)^1/x\}$, le premier terme meurt par famine dans l'état $x\{(\lambda z.z)^0/x\}$ tandis que le second donne une valeur [1], à savoir $\lambda y.xy^\infty\{(\lambda z.z)^0/x\}$ (fonction en attente d'un argument pour mourir).
Plus récemment, une autre version avec ressources finies fut étudiée, qui tira de son comportement calculatoire le nom de λ-calcul différentiel [ER03], mais qui finalement est à peu près équivalente. Nous nous placerons ici dans le cadre du λ-calcul dit avec ressources [PT09], version la plus récente, qui comporte des arguments linéaires et infinis.

1.2 Syntaxe

La différence avec le λ-calcul usuel est donc que désormais, le deuxième opérande d'une application n'est plus un argument unique, mais un paquet d'arguments, qui sont eux-mêmes disponibles de façon linéaire ou infinie. Nous nous appuierons ici sur la syntaxe présentée ci-après, développée par Paolo Tranquilli et Michele Pagani [PT09].

Λ :	M, N, L	$::= x \mid \lambda x.M \mid (MP)$	termes
Λ^{arg} :	$M^{(!)}, N^{(!)}$	$::= M \mid M^!$	arguments
Λ^b :	P, Q, R	$::= [M_1^{(!)}, \dots, M_n^{(!)}]$	paquets [2]
$\Lambda^{(b)}$:	A, B	$::= M \mid P$	expressions
$\mu, \nu \in \mathbb{N}\langle\Lambda\rangle$	$\pi, \rho \in \mathbb{N}\langle\Lambda^b\rangle$	$\alpha, \beta, \gamma \in \mathbb{N}\langle\Lambda^{(b)}\rangle := \mathbb{N}\langle\Lambda\rangle \cup \mathbb{N}\langle\Lambda^b\rangle$	sums

(a) Grammaire des termes, paquets, expressions, sommes

$$\begin{array}{rclcrcl} \lambda x.(\sum_i M_i) & := & \sum_i \lambda x.M_i & \quad & [(\sum_i M_i)] \cdot P & := & \sum_i [M]_i \cdot P \\ (\sum_i M_i)P & := & \sum_i M_i P & & [(\sum_i M_i)^!] \cdot P & := & [M_1^!, \dots, M_k^!] \cdot P \\ M(\sum_i P_i) & := & \sum_i M P_i & & & & \end{array}$$

(b) Notations on $\mathbb{N}\langle\Lambda^{(b)}\rangle$.

FIGURE 1 – Syntaxe du calcul avec ressources

L'ensemble Λ^b correspond aux paquets d'arguments, arguments disponibles en quantité infinie ($M^!$) ou linéaire (M). Les termes Λ sont alors construits en prenant des paquets comme second argument d'une application, exprimant le caractère éventuellement limité des ressources. La présence potentielle de plusieurs arguments dans un paquet induit un choix à faire lors de la consommation, et donc du non-déterminisme, d'où les sommes.

1. *En faisant dans cet exemple de l'évaluation paresseuse*
2. *Ce sont des multiensembles,* 1 *étant le multiensemble vide, et* $\cdot$ *l'opérateur*

$$y\langle N/x\rangle := \begin{cases} N & \text{si } y = x, \\ 0 & \text{sinon,} \end{cases} \qquad (\lambda y.L)\langle N/x\rangle := \lambda y.(M\langle N/x\rangle), \quad y \notin \mathrm{FV}(N) \cup \{x\},$$
$$(MP)\langle N/x\rangle := M\langle N/x\rangle P + M(P\langle N/x\rangle)$$
$$[M]\langle N/x\rangle := [M\langle N/x\rangle], \qquad 1\langle N/x\rangle := 0,$$
$$[M^!]\langle N/x\rangle := [M\langle N/x\rangle, M^!] \qquad (P \cdot R)\langle N/x\rangle := P\langle N : x\rangle \cdot R = P \cdot R\langle N/x\rangle,$$

(a) Substitution linéaire

$$A\langle\langle N^{(!)}/x\rangle\rangle := \begin{cases} A\langle N/x\rangle & \text{si } N^{(!)} = N, \\ A\{x + N/x\} & \text{si } N^{(!)} = N^!, \end{cases}$$
$$A\langle\langle [N_1^{(!)}, \ldots, N_k^{(!)}]/x\rangle\rangle := A\langle\langle N_1^{(!)}/x\rangle\rangle \ldots \langle\langle N_k^{(!)}/x\rangle\rangle, \quad x \notin \bigcup_{i=1}^{k} \mathrm{FV}(N_i^{(!)})$$

NB : l'ordre dans lequel on choisit les arguments dans le paquet ne change pas le résultat.

(b) Substitutions des arguments et des paquets.

FIGURE 2 – Règles de substitutions pour les termes, arguments et paquets

Les règles de substitutions sont adaptées aux différents types d'arguments. La substitution usuelle du λ-calcul est ici notée $t\{N/x\}$. Comme celle-ci ne contient intrinsèquement aucune information sur le nombre d'utilisations à venir de N, elle ne peut intervenir que dans le cas de ressources infinies. On définit une substitution linéaire $t\langle N/x\rangle$, qui capture l'idée qu'on doit se servir de N une et une seule fois, et enfin la substitution des paquets $t\langle\langle [N_1, \ldots, N_k]/x\rangle\rangle$.

Il est clair que l'on peut facilement plonger le λ-calcul usuel dans le λ-calcul avec ressources par la transformation $t \mapsto t^*$ suivante : $\begin{cases} x^* & = & x \\ (\lambda x.t)^* & = & \lambda x.(t^*) \\ (tu)^* & = & (t^*)[(u^*)^!] \end{cases}$

1.3 Réduction

Il ne reste plus qu'à définir la β-réduction pour ce système. On commence par définir de façon générale les règles de passage au contexte pour une relation.

$$\frac{}{x \longrightarrow_R x}\ \texttt{var} \qquad \frac{t \longrightarrow_R u}{tB \longrightarrow_R uB}\ \overline{@}_l \qquad \frac{B \longrightarrow_R B'}{tB \longrightarrow_R tB'}\ \overline{@}_r$$

$$\frac{}{1 \longrightarrow_R 1}\ \texttt{bag1} \qquad \frac{b \longrightarrow_R b'}{[b] \cdot B \longrightarrow_R [b'] \cdot B}\ \overline{\texttt{bag}\ell} \qquad \frac{b \longrightarrow_R b'}{[b^!] \cdot B \longrightarrow_R [b'^!] \cdot B}\ \overline{\texttt{bag!}}$$

$$\frac{t \longrightarrow_R u}{\lambda x.t \longrightarrow_R \lambda x.u}\ \overline{\lambda} \qquad \frac{t_i \longrightarrow_R t'_i \qquad \forall j \neq i,\ t'_j = t_j}{\sum_j t_j \longrightarrow_R \sum_j t'_j}\ \texttt{sum}$$

FIGURE 3 – Passage au contexte d'une relation $\longrightarrow_R$

Définition 1 (β**-réduction**)**.** La relation de β-réduction (dite à grands pas) est la plus petite relation satisfaisant les règles du passage au contexte et la règle suivante :

$$\frac{}{(\lambda x.M)P \longrightarrow_\beta M\langle\langle P/x\rangle\rangle\{0/x\}}\ \mathtt{g}$$

Nous allons par la suite être naturellement amenés à considérer d'autres réductions plus faibles, définies ci-après.

Définition 2 (**Réduction externe**)**.** La *réduction externe* est la plus petite relation satisfaisant `g` et toutes les règles de passage au contexte exceptée $\overline{\texttt{bag!}}$.

Définition 3 (**Réduction de tête**)**.** La *réduction de tête* est la plus petite relation satisfaisant `g`, `var`, $\overline{@}_l$, $\overline{\lambda}$ et `sum`.

Définition 4 (**Réduction de tête faible**)**.** La *réduction de tête faible* est la relation satisfaisant uniquement `g`, `var`, $\overline{@}_l$ et `sum`.

Il est à noter que l'on dispose dans ce système de plusieurs résultats, notamment sur la confluence, la terminaison dans le cas typé, et caetera. [ER03][PT09]

2 Une machine abstraite pour le calcul avec ressources

2.1 La machine de Krivine usuelle

Introduite par Krivine [Kri04], le but de cette machine abstraite est d'effectuer une sorte de compilation d'un λ-terme. La machine est constituée d'un terme de tête t et son environnement ρ (ce que l'on nomme une clôture), devant une pile π de clôtures de même type.

$Push$:	$t\ u, \rho * \pi$	$\to$	$t, \rho * (u, \rho) :: \pi$
$Grab$:	$\lambda x.t, \rho * (u, \rho') :: \pi$	$\to$	$t, [x \mapsto (u, \rho')] \cdot \rho * \pi$
$Access1$:	$x, [y \mapsto (u, \rho')] \cdot \rho * \pi$	$\to$	$x, \rho * \pi$
$Access2$:	$x, [x \mapsto (u, \rho')] \cdot \rho * \pi$	$\to$	$u, \rho' * \pi$

FIGURE 4 – La machine de Krivine usuelle

Quitte à retraduire[3] (en dépilant) un état de la machine vers les λ-termes, on obtient le résultat suivant :

Proposition 5 ([Lan07])**.** *La machine de Krivine réalise exactement la réduction de tête faible.*

De plus, on peut également poursuivre l'exécution sous les λ (par exemple en définissant une sortie et en y pré-inscrivant en tête les λ en question), ce qui permet d'obtenir une forme normale de tête. Ceci est intéressant dans la mesure où par exemple on sait que, dans le λ-calcul ordinaire, les termes solvables sont exactement les termes admettant une forme normale de tête [Kri90]. Le résultat obtenu peut également être interprété en termes d'arbres de Böhm [ER05].

3. *Par induction :* - traduction(x,[x↦ (u,ρ')]·ρ)=traduction(u,ρ'),
- traduction(t u,ρ)=traduction(t,ρ)traduction(u,ρ),
et ainsi de suite..

2.2 Idées générales

L'objectif est donc ici d'adapter la machine de Krivine à l'exécution de λ-termes avec ressources. Les deux principales difficultés sont de ne pas dupliquer les ressources linéaires, lors d'un *push* notamment, et la présence de sommes de termes, liées au non-déterminisme. Pour ce dernier problème, on peut voir l'exécution d'un terme comme l'exploration d'un arbre des possibles, chaque terme de la somme correspondant à une possibilité. Cela oblige cependant à travailler en quelque sorte sur des sommes d'états machines.

Pour la non-duplication des ressources, deux solutions sont envisageables : soit l'on partage définitivement les ressources linéaires de l'environnement lors du *push*, et dans ce cas c'est là qu'intervient le non-déterminisme ; soit l'on travaille avec un environnement global, et des environnements locaux jouant uniquement le rôle de pointeurs. C'est cette dernière version que je présenterai principalement ici.

Il est intéressant de voir, *a priori*, à quoi correspondent dans ce cadre-là les morts par famine (manque d'arguments) et abondance (trop d'arguments linéaires). En effet, si la famine revient à rechercher une ressource dans un paquet vide, l'abondance correspond à la présence dans l'environnement de ressources linéaires sur lesquelles plus personne ne pointe. Ce qui peut amener, lors de l'implémentation, à effectuer un travail de gestion de la mémoire, avec un comptage de références.

2.3 Ladite machine

On définit en premier lieu les notions de clôture, pile et environnement.

$C ::= M, n$	*clôture*
$\pi ::= C :: \pi$	*pile*
$E_n ::= x, C, n' \mid \emptyset$	*environnement courant*
$E ::= (\emptyset, E_1, E_2 \dots E_n \dots)$	*environnement global*

FIGURE 5 – Grammaire de la nouvelle machine abstraite

L'entier m d'une clôture joue intuitivement le rôle d'un pointeur vers E, c'est ce qui va permettre d'effectuer du partage de ressources. Pour une clôture de tête t, n, on parcourt un environnement de la façon suivante :

$$\text{si } E_n = x, C, n_1,\ E_{n_1} = x_1, C_1, n_2\ \dots\ E_{n_k} = x_k, C_k, 0 \text{ (et } E_0 = \emptyset)$$
$$\text{alors } E(n) \equiv [x_1 \mapsto C_1] \cdot [x_2 \mapsto C_2] \cdots [x_k \mapsto C_k].$$

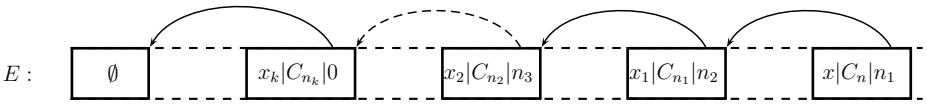


FIGURE 6 – Environnements

L'idée naturelle aurait peut-être été de pointer directement sur des paquets, mais dans ce cas, un environnement aurait été constitué d'une suite de pointeur, qu'il

eût fallu dupliquer dans la pile, et la taille des objets aurait beaucoup grossie, ce qui est dommage en faisant du partage de ressources.
Un état de la machine est la donnée d'un terme de tête et son environnement courant, devant une pile, dans un environnement global :

$$t, m * \pi * E$$

Du fait du non-déterminisme, on peut avoir à étudier une somme de termes pris dans des environnements différents. Naturellement, on a envie de tester pour chaque terme. On considère donc des machines sous la forme suivante [4] [5] :

$$\kappa \equiv \sum (t, m * \pi * E)$$

Push :	$t\ B, m * \pi * E$	$\longrightarrow_\kappa$	$t, m * (B, m) :: \pi * E$
Grab :	$\lambda x.t, m * (U, n) :: \pi * E$	$\longrightarrow_\kappa$	$t, m' * \pi * E'$ $avec \begin{cases} m' \text{ frais} \\ E'(m') = x, (U, n), m \\ E'(m) = E(m) \quad \forall m \neq m' \end{cases}$
Access1 :	$x, m * \pi * E$ $où\ E(m) = y, (B, n), m'$	$\longrightarrow_\kappa$	$x, m' * \pi * E$
Access2 :	$x, m * \pi * E$ $où\ E(m) = x, (B, n), m'$	$\longrightarrow_\kappa$	$\sum_{b^{(!)} \in B} (b, n * \pi * E_b)$ $avec \begin{cases} E_b(m) = x, B \backslash \{b^{(!)}\}, n, m' \\ E_b(i) = E(i) \quad \forall i \neq m \end{cases}$

FIGURE 7 – Pas d'exécution de la nouvelle machine de Krivine

Les règles d'exécution sont basées sur les règles de la machine de Krivine usuelle. Le *push* ne présente aucune nouveauté, ni le *grab* (modulo la construction explicite du petit bout d'environnement désiré). En revanche, l'*access* doit changer. D'une part, il doit permettre de détecter la famine, cas où l'on pointe sur un paquet vide, et dans la même optique, on doit marquer dans l'environnement global la consommation effective des ressources (lorsqu'elles sont linéaires). On définit pour cela $B \backslash \{b^{(!)}\}$ de façon naturelle : $\begin{cases} B \backslash \{b^!\} = B \\ (a \cdot B) \backslash \{b\} = a \cdot (B \backslash \{b\}) \\ (b \cdot B) \backslash \{b\} = B \end{cases}$.
D'autre part, on doit tester toutes les possibilités de valeurs, d'où la somme, c'est ici qu'intervient le non-déterminisme.

Il faut encore rajouter à cela l'inférence naturelle permettant de passer d'une somme d'états possibles à une autre, via une réduction dans l'une des branches :

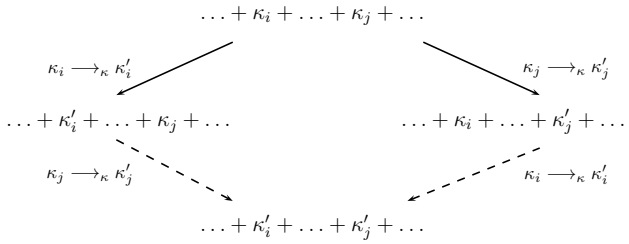
$$\frac{\kappa_i \longrightarrow_\kappa \kappa'_i \qquad \forall j \neq i,\ \kappa'_j = \kappa_j}{\sum_j \kappa_j \longrightarrow_\kappa \sum_j \kappa'_j}$$

4. *La somme est à prendre au sens de multiset, puisque l'on veut avoir accès à chaque terme individuellement*

5. *NB : on a facilement* $\{\sum(t, m * \pi * E)\} \subset \{\sum(t, m) * \sum(\pi) * \sum E\}$*, quitte à modifier la grammaire, on pourrait décréter que tout s'exécute dans une machine plus usuelle, avec une seule pile, mais on n'y comprendrait plus rien...*

La stratégie d'exploration des branches est, bien entendu, libre, et relève uniquement de la technique d'implémentation. Lors de ce stage, la machine a été implémentée avec pour stratégie un parcours en profondeur de l'arbre (*i.e.* on teste jusqu'à aboutir à la possibilité courante, puis on remonte jusqu'au nœud le plus proche avant de continuer). De plus, le fait de fonctionner par exploration de branches permet de ne pas avoir à garder en mémoire un arbre trop important. En effet, il suffit de conserver la branche courante, et l'arborescence au dessus.

On note au passage que $\longrightarrow_\kappa$ a clairement la propriété du diamant : le seul moment où intervient un éventuel choix est celui du choix d'une branche à explorer.



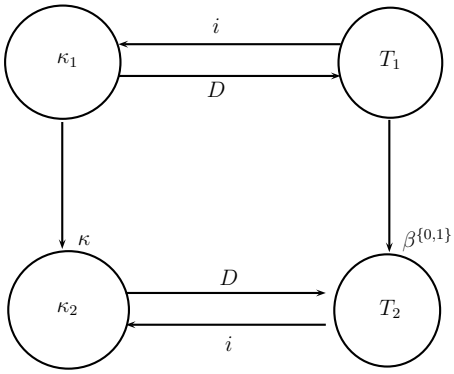
Or, clairement, si l'on en choisit une puis une autre, en faisant l'inverse on obtient rigoureusement la même chose, puisque les deux n'entrent aucunement en interaction. D'où le résultat suivant :

Proposition 6 (**Confluence**). $\longrightarrow_\kappa$ *est confluente*

3 Correction de la machine

3.1 Schéma de principe

Deux choses sont à prouver : d'une part, que les pas d'exécution de la machine peuvent être reliés en un certain sens à des β-réductions, et d'autre part, que la machine s'arrête bien sur un terme en forme normale de tête. On peut néanmoins aussi s'intéresser au contrôle plus en détail de l'exécution, auquel cas il est utile de pouvoir montrer dans l'autre sens que l'on sait construire une suite de pas machine correspondant à une substitution.



Il est à noter que si l'on espère avoir $i \circ D = id_{\lambda_t}$, il est certain que l'on aura $D \circ i \neq id_\kappa$

FIGURE 8 – Principe de la décompilation

On a relativement facilement, par la fonction $i : \begin{cases} \lambda_t \mapsto (\lambda_t, 0 * [], \emptyset) \\ \sum(\lambda_t) \mapsto \sum(i(\lambda_t)) \end{cases}$ un moyen d'injecter un λ-terme dans les états machines. On cherche donc, désormais, à construire une fonction $D : \kappa \to \Lambda^{(b)}$ de décompilation, qui puisse, à chaque état machine, faire correspondre un λ-terme. C'est la partie horizontale de la Figure 8.
Une fois ces fonctions construites, l'objectif est donc de montrer qu'à tout pas $\kappa_1 \longrightarrow_\kappa \kappa_2$ correspond un pas $D(\kappa_1) \xrightarrow{\{0,1\}}_\beta D(\kappa_2)$.

3.2 Fonctions de traduction/décompilation

À l'inverse de la machine de Krivine usuelle, du fait du partage de ressources, la décompilation doit être effectuée soigneusement, dans un certain ordre défini. Si la structure du λ-calcul nous pousse à raisonner par induction, il est néanmoins clair que l'on ne peut ici construire nos fonctions de façon purement inductive. En effet, l'exemple $\kappa \equiv x, n * ([x], n) :: \pi * E, où\ E_n = x, [\alpha^1], \cdot, \cdot$ montre bien pourquoi on ne peut poser $D(C_1 * C_2) = D(C_1)D(C_2)$. L'idée est donc, un peu à la manière d'une continuation, de transmettre et faire évoluer l'environnement tout au long de la traduction.

$D : \kappa \to \mathbb{N}\langle\Lambda\rangle$	*la décompilation*
$\mu : \Lambda * \mathbb{N} * env \to \mathbb{N}\langle\Lambda * env\rangle$	*la passe de continuation*
$\mu^! : \Lambda * \mathbb{N} * env \to \mathbb{N}\langle\Lambda^b * env\rangle$	
$\delta : env \mapsto \{0, 1\}$	*vérifie la non-abondance*

$$D(\kappa) = \sum(\delta(E).t),\ où\ \sum(t, E) = \mu(\kappa)$$

FIGURE 9 – Fonctions de décompilation

On définit donc deux fonctions, μ et $\mu^!$ (*Figure 10*). En effet, on a fondamentalement besoin de deux comportements différents :
- une recherche linéaire (μ) : on veut une ressource et une seule, typiquement dans le cas du premier opérande d'une application ou ressource linéaire dans un paquet
- une recherche exponentielle ($\mu^!$) : on veut tout ce que l'on peut avoir, dans le cas où au départ on part d'une ressource infinie dans un paquet.

De même que pour les pas d'exécution de la machine, on ne duplique pas les ressources, et l'on véhicule tout du long un environnement que l'on modifie quand on retire des ressources linéaires. Du coup, cette espèce de passe d'environnement définit un ordonnancement dans la décompilation.

Propriété 7. On a :
(i) $\forall t$ clos, $\mu(t, \cdot, E) = t, E$ et $\mu^!(t, \cdot, E) = [t^!], E$
(ii) $\forall t, u$, pour un environnement E, on obtient les mêmes résultats en calculant d'abord $\mu(t, n, E)$ puis $\mu(u, m, E)$ qu'en faisant l'inverse.

La première propriété se prouve facilement par induction sur t. La seconde est plus intéressante, elle exprime le fait que l'on eut pu de façon indifférenciée

$$
\mu : \left\{
\begin{array}{lcl}
x,m,E & \mapsto & \left\{
\begin{array}{ll}
\mu(x,m',E) & si\ E_m = y \neq x,\cdot,\cdot,m' \\
0,E & si\ E_m = x,[],\cdot,\cdot \\
x,E & si\ E_m = x,\$,\cdot,\cdot \\
\sum\limits_{b^{(!)}\in B} \mu(b,m',E^b) & si\ E_m = x,B,m',\cdot \\
\text{où } E^b_m = x, B\backslash b^{(!)} &
\end{array}
\right. \\
b\cdot B,m,E & \mapsto & \sum\limits_{i\in I}\sum\limits_{j\in J_i}(b_i\cdot B_{ij},E_{i,j}) \quad \text{où} \left|
\begin{array}{l}
\sum\limits_{i\in I}(b_i,E_i) = \mu(b,m,E) \\
\sum\limits_{j\in J_i}(B_{ij},E_{ij}) = \mu(B,m,E_i)
\end{array}
\right. \\
b^!\cdot B,m,E & \mapsto & \sum\limits_{i\in I}\sum\limits_{j\in J_i}(B_i\cdot B_{ij},E_{i,j}) \quad \text{où} \left|
\begin{array}{l}
\sum\limits_{i\in I}(B_i,E_i) = \mu^!(b,m,E) \\
\sum\limits_{j\in J_i}(B_{ij},E_{ij}) = \mu(B,m,E_i)
\end{array}
\right. \\
\lambda x.t,m,E & \mapsto & \sum(\lambda x.t',E') \text{ où } \sum(t',E') = \mu(t,p,[x\mapsto(\$,m),m]\cdot E) \\
tB,m,E & \mapsto & \sum(t''B',E'') \text{ où } \left|
\begin{array}{l}
\sum t'',E'' = \mu(t,m,E') \\
\sum(B',E') = \mu(B,E)
\end{array}
\right.
\end{array}
\right.
$$

NB : le \$ sert à marquer les constantes, notamment lorsque l'on passe sous un λ...

$$
\mu^! : \left\{
\begin{array}{lcl}
x,m,E & \mapsto & \left\{
\begin{array}{ll}
\mu^!(x,m',E) & si\ E_m = y \neq x,\cdot,\cdot,m' \\
1,E & si\ E_m = x,[],\cdot,\cdot \\
\left[x^!\right],E & si\ E_m = x,\$,\cdot,\cdot \\
\sum\limits_{i\in I}\sum\limits_{j\in J_i}([P_i]\cdot B_{ij},E_{ij})) + \mu^!(B,m',E) & \\
\quad \text{où} \left|
\begin{array}{l}
\sum\limits_{i}(P_i,E_i) = \mu(b,m',E^b) \\
\sum\limits_{j\in J_i} B_{ij},E_{ij} = \mu^!(B,m',E_i)
\end{array}
\right. & si\ E_m = x,b\cdot B,m',\cdot \\
\sum\limits_{i\in I}\sum\limits_{j\in J_i}([P_i]\cdot B_{ij},E_{ij})) + \mu^!(B,m',E) & \\
\quad \text{où} \left|
\begin{array}{l}
\sum\limits_{i}(P_i,E_i) = \mu^!(b,m',E) \\
\sum\limits_{j\in J_i} B_{ij},E_{ij} = \mu^!(B,m',E_i)
\end{array}
\right. & si\ E_m = x,b^!\cdot B,m',\cdot
\end{array}
\right. \\
b^{(!)}\cdot B,m,E & \mapsto & \sum\limits_{i\in I}\sum\limits_{j\in J_i}(b_i\cdot B_{ij},E_{i,j}) \quad \text{où} \left|
\begin{array}{l}
\sum\limits_{i\in I}(b_i,E_i) = \mu^!(b,m,E) \\
\sum\limits_{j\in J_i}(B_{ij},E_{ij}) = \mu^!(B,m,E_i)
\end{array}
\right. \\
\lambda x.t,m,E & \mapsto & \sum\limits_{i}[\lambda x.b_{i,1},\ldots,\lambda x.b_{i,p_i}],E_i \quad \text{où} \quad \sum\limits_{i}[b_{i,1},\ldots,x.b_{i,p_i}],E_i = \mu^!(t) \\
tB,m,E & \mapsto & \sum\limits_{i,j}(t_{i,j}B_i,E_{i,j}) \text{ où } \left|
\begin{array}{l}
\sum\limits_{i,j}(t_{i,j},E_{i,j}) = \mu^!(t,m,E_i) \\
\sum\limits_{i}(B_i,E_i) = \mu^!(B,E)
\end{array}
\right.
\end{array}
\right.
$$

$$
\mu(C_1...C_n * E) = \mu(C_1...C_{n-1} * E')\ B_n \quad \text{où } B_n, E' = \mu(C_n * E)
$$

FIGURE 10 – Définition de μ et $\mu^!$

choisir de décompiler en premier lieu les choses à gauche. Intuitivement, il suffit de voir la traduction comme un parcours des variables libres, chacune amenant à piocher des ressources dans l'environnement. Formellement, on peut définir l'ensemble des ressources disponibles dans l'environnement, et voir que l'on peut les enlever dans n'importe quel ordre, ça ne change rien.

3.3 La preuve

Proposition 8. *Si* $\kappa_1 \underset{push}{\longrightarrow} \kappa_2$, *alors* $D(\kappa_1) = D(\kappa_2)$

Démonstration. Par définition. □

Proposition 9. *Si* $\kappa_1 \underset{access}{\longrightarrow} \kappa_2$, *alors* $D(\kappa_1) = D(\kappa_2)$

Démonstration. Si l'on a fait un pas avec *access*, alors on avait en tête une variable. Or μ restreinte à une variable de tête et *access* fonctionnent exactement pareil. Donc rien ne change, le *access* ne fait que gagner une étape de traduction. □

Pour le dernier pas, on a besoin de lemmes techniques, afin d'expliciter un peu plus la décompilation. On commence par montrer que l'on peut s'intéresser aux variables libres dans un ordre quelconque, et en particulier en fixer une, et ne la remplacer qu'après.

Lemme 10. $D(t * \pi * [x \mapsto B] \cdot E) = D(D(t * \pi * [x \mapsto \$] \cdot E) * [x \mapsto B])$ [6]

Démonstration. On raisonne par induction sur t.

- Les deux cas atomiques ($x, y \neq x$) sont triviaux.
- Si $t \equiv \lambda y.t$

$$\begin{aligned} D(\lambda y.t * [x \mapsto B]) &= \lambda y.(D(t * [y \mapsto \$] \cdot [x \mapsto B])) \\ \textit{(par induction)} &= \lambda y.(D(D(t * \pi * [x \mapsto \$] \cdot [y \mapsto \$]) * [x \mapsto B])) \\ (\textit{car } y \notin \mathrm{FV}(B)) &= D(\lambda y.D(t * \pi * [x \mapsto \$] \cdot [y \mapsto \$]) * [x \mapsto B]) \\ &= D(D(\lambda y.t * \pi * [x \mapsto \$]) * [x \mapsto B]) \end{aligned}$$

On considère y comme une constante dans D(t...), quitte à la substituer par une valeur et effectuer ensuite la substitution inverse, clairement, cela ne change rien.

- Si $t \equiv tU$, regardons plus en détail ce que cela signifie

D'une part, pour $D(tU * \pi * [x \mapsto B] \cdot E)$:

on calcule $\mu(U * [x \mapsto \$] \cdot E)$, ce qui par induction, revient à calculer $\mu(U * [x \mapsto \$] \cdot E)$ qui nous donne $\sum_i U_i, E_i$, puis $\mu(U_i * [x \mapsto B] \cdot E_i)$, dont on obtient $\sum_j U_{i,j}, E_{i,j}$.

Puis pour $(t, E_{i,j})$, on commence par $\mu(t * [x \mapsto \$] \cdot E_{i,j}) = \sum_k t_k, E_{i,j,k}$ et enfin on calcule $\mu(t_k * [x \mapsto B] \cdot E_{i,j,k})$

D'autre part, pour $D(D(tU * \pi * [x \mapsto \$] \cdot E) * [x \mapsto B])$:

on calcule $\mu(tU * [x \mapsto \$] \cdot E)$, ce qui revient à calculer $\mu(U * [x \mapsto \$] \cdot E)$ qui nous donne $\sum_i U_i, E_i$, puis $\mu(t * [x \mapsto \$] \cdot E_i)$.

6. *Il faudrait en théorie écrire les choses sur μ et non D, mais cela alourdirait considérablement les notations, et il n'y a ici aucune ambiguïté sur l'ordonnancement, par abus, nous conserverons donc cette version-là*

dont on obtient $\sum_j t_{i,j}, E_{i,j}$. Puis pour $(t_{i,j}U_i, E_{i,j})$, on commence par $\mu(U_i * [x \mapsto \$] \cdot E_{i,j}) = \sum_k U_{i,j,k} * E_{i,j,k}$ et enfin $\mu(t_{i,j} * [x \mapsto B] \cdot E_{i,j,k})$

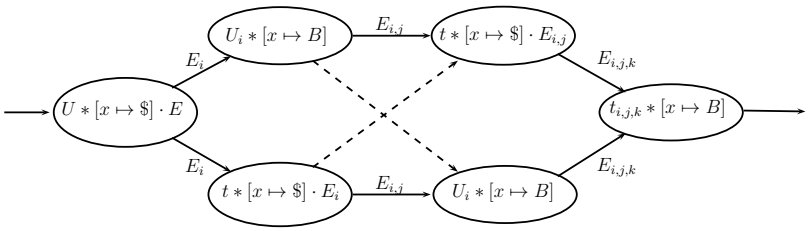


FIGURE 11 – Ce qu'il se passe

Au final, on ne fait qu'inverser les deux étapes centrales (*cf. Figure 11*), ce qui revient exactement à montrer l'équivalence des stratégies de décompilation droite-gauche / gauche-droite (évidemment cruciale, montrée lors de la construction de μ (*Propriété 7*)) □

Disposant du lemme précédent, il ne nous reste qu'à montrer que, pour une variable donnée, l'associer à un paquet d'arguments correspond bien à faire la substitution dans le λ-terme. Pour cela, nous aurons besoin du résultat suivant :

Lemme 11. *Soit $B = [b_1^{(!)}, \ldots, b_n^{(!)}]$ un paquet d'arguments. On note $A \lhd B$ si et seulement si $\exists I \subset [\![1; n]\!]$ tel que $A = \{b_i, i \in I\}$ et $A \cap \Lambda^! = B \cap \Lambda^!$.*
Soit donc $A \lhd B$, on note $\bar{A}^B$ l'unique C tel que $\begin{cases} A \cap C = B \cap \Lambda^! \\ A \cdot C = B \end{cases}$ (son existence et son unicité sont immédiates).

Alors,

$$(tU)\langle\langle B/x\rangle\rangle = \sum_{A \lhd B} (t\langle\langle A/x\rangle\rangle)(U\langle\langle \bar{A}^B/x\rangle\rangle)$$

Démonstration. Par récurrence sur la taille de B. □

On peut alors montrer le lemme attendu :

Lemme 12. *Si $FV(t) \subset \{x\}$ et $FV(B) = \emptyset$, alors $D(t, 1 * [] * E \equiv [x \mapsto B, 0]) = t\langle\langle B/x\rangle\rangle\{0/x\}$*

Démonstration. On raisonne par induction sur la taille de B.
• Si $B = [b]$:
alors, par induction sur t :

- si $t \equiv y \neq x$, en considèrant donc y comme une constante (sinon $y \in FV(t)$) [7], on a $y\langle\langle [b]/x\rangle\rangle\{0/x\} = y\langle b/x\rangle = 0$ et $D(y, 1 * [] * E : [x \mapsto [b]; y \mapsto \$]) = \delta([b]) \cdot y = 0$ car $[b]$ n'est pas utilisé
- si $t \equiv x$, $x\langle\langle [b]/x\rangle\rangle = b$ et $D(x, *[] * [x \mapsto [b]]) = b$ par définition
- si $t \equiv \lambda y.t$, par induction, en considérant y comme une constante dans t [8], on a $D(t, 1 * [] * E : [x \mapsto [b]]) = t\langle\langle [b]/x\rangle\rangle\{0/x\}$

7. *En fait ce cas n'intervient pas dans l'énoncé du lemme, mais on peut y être confronté par induction*
8. *Ou n'importe quoi d'autre en substitution, notamment un terme clos*

d'où $D(\lambda y.t, 1 * [] * E : [x \mapsto [b]]) = \lambda y.D(t, 2 * [] * E : [y \mapsto \$, 0; x \mapsto [b], 0])$
$= \lambda y.D(t, 1 * [] * E : [x \mapsto [b]])$
$= \lambda y.(t\langle\langle [b]/x\rangle\rangle\{0/x\})$
$= (\lambda y.t)\langle\langle [b]/x\rangle\rangle\{0/x\}$

On remarque que ceci reste valable pour $b^!$.
- si $t \equiv tU$, on a $T = (tU)\langle\langle [b]/x\rangle\rangle\{0/x\} = ((t\langle b/x\rangle))U\{0/x\} + t(U\langle b/x\rangle)\{0/x\}$.

De plus, - si $x \in FV(U)$: -si $x \in FV(t)$: $D(tU, ...) = 0 = T$ (mort par famine)
-sinon, on a $T = t(U\langle b/x\rangle)\{0/x\}$ et
$D(tU, ...) = tD(U...)$, par induction sur U, on en déduit ce que l'on voulait
-sinon, on a $T = (t\langle b/x\rangle)U\{0/x\}$ et $D(tU, ...) = D(t...)U$, même chose par induction sur t

- Si $B = [b^!]$:

alors, par induction sur t :
- si $t \equiv y \neq x$, en considèrant donc y comme une constante (sinon $y \in FV(t)$), on a $y\langle\langle [b^!]/x\rangle\rangle\{0/x\} = y\{b/x\} = y$ et $D(y, 1 * [] * E : [x \mapsto [b^!]; y \mapsto \$]) = \delta([b^!]) \cdot y = y$
- si $t \equiv x$, $x\langle\langle [b^!]/x\rangle\rangle\{0/x\} = b$ et $D(x, *[] * [x \mapsto [b^!]]) = b$ par définition
- si $t \equiv \lambda y.t$, on procède comme pour $[b]$
- si $t \equiv tU$, on a $T = (tU)\langle\langle [b^!]/x\rangle\rangle = (t\{b/x\})(U\{b/x\})$.

Or $D(tU, 1 * [] * E \equiv [x \mapsto b^!, 0]) \equiv D(t, 1 * [] * E \equiv [x \mapsto b^!, 0]) \cdot D(U, 1 * [] * E \equiv [x \mapsto b^!, 0])$, car $\mu(U, ... * E)$ laisse E inchangé, la seule ressource étant $b^!$. Par induction sur t et U, on en déduit le résultat escompté.

- Si $B_0 \equiv b^{(!)} \cdot B$, on a $t\langle\langle b^{(!)} \cdot B/x\rangle\rangle = t\langle\langle b^{(!)}/x\rangle\rangle\langle\langle B/x\rangle\rangle$

- si $t \equiv y \neq x$, on a $D(y, 1 * [] * E \equiv [x \mapsto b^{(!)} \cdot B; y \mapsto \$])$
$= \delta(b^{(!)} \cdot B) \cdot D(y, 1 * [] * \emptyset])$
$= \delta(b^{(!)} \cdot B) \cdot y$
$= y\langle\langle b^{(!)} \cdot B/x\rangle\rangle$
- si $t \equiv x$, on a $D(x, 1 * [] * E \equiv [x \mapsto b^{(!)} \cdot B, 0])$
$= \delta(b^{(!)}) \cdot D(x, 1 * [] * E \equiv [x \mapsto B, 0]])$
$+ \delta(B) \cdot D(x, 1 * [] * E \equiv [x \mapsto b^{(!)}, 0]])$
$= \delta(b^{(!)}) \cdot x\langle\langle B/x\rangle\rangle + \delta(B) \cdot x\langle\langle b^{(!)}/x\rangle\rangle$ *(par induction)*
$= x\langle\langle b^{(!)} \cdot B/x\rangle\rangle$.
- si $t_0 \equiv \lambda y.t$, on a par induction $D(t, 2 * [] * E : [y \mapsto \$, 0; x \mapsto B_0]) = t\langle\langle B_0/x\rangle\rangle\{0/x\}$
D'où $D(\lambda y.t, 1 * [] * E : [x \mapsto B_0]) = \lambda y.D(t, 2 * [] * E : [y \mapsto \$, 0; x \mapsto B_0, 0])$
$= \lambda y.(t\langle\langle B_0/x\rangle\rangle\{0/x\})$
$= (\lambda y.t)\langle\langle B_0/x\rangle\rangle\{0/x\}$ $(y \notin \mathrm{FV}(B_0))$
- si $t_0 \equiv tU$, en notant $\lambda x.\mu(\kappa)B$ pour $\sum_i(\lambda x.t_i B, E_i)$ *où* $\mu(\kappa) = \sum_i(t_i, E_i)$,

on a $\mu(tU * E \equiv [x \mapsto B_0]) = \sum_i \mu(t * E_i \equiv [x \mapsto B_i])U_i$.

Or on a facilement, en posant $I_k = \{i / B_i = B_k\}$, $\sum_{i \in I_k} U_i = U\langle\langle \bar{B_k}^{B_0}/x\rangle\rangle\{0/x\}$
(intuitivement, lorsqu'on laisse le même environnement, on a consommé les mêmes ressources linéaires). Ce qui donne, en regroupant les termes :
$\mu(tU * [x \mapsto B_0]) = \sum_{B_k \subset B_0} \mu(t * E_k \equiv [x \mapsto B_k])U\langle\langle \bar{B_k}^{B_0}/x\rangle\rangle\{0/x\}$

$$= \sum_{B_k \subset B_0} (t\langle\langle B_k/x\rangle\rangle\{0/x\})U\langle\langle \bar{B_k}^{B_0}/x\rangle\rangle\{0/x\} \qquad (induction)$$
$$= (tU)\langle\langle B_0/x\rangle\rangle\{0/x\} \qquad (lemme\ 11)$$

□

On peut se ramener au cas général ($FV(B) \neq \emptyset$) puisque, comme c'est le dernier appel à l'environnement, toutes les ressources non utilisées généreront une mort par abondance, donc les seul termes intéressants sont ceux dans lesquels μ a bien été chercher au fond de l'environnement toutes les ressources de B.

Proposition 13. *Si* $\kappa_1 \underset{grab}{\longrightarrow} \kappa_2$, *alors* $D(\kappa_1) \xrightarrow{1}_\beta D(\kappa_2)$

Démonstration. On se restreint, sans perte de généralité, à κ_1 de la forme $\lambda x.t, m * B, n * E$, avec un seul terme sur la pile. L'énoncé devient $D(\lambda x.t, m * B, n * E) \longrightarrow_\beta D(t, p * [] * E')$ avec $E' \equiv E_p = (x, B, n, m) \cup E$.
Par application des lemmes précédents, on obtient [9] :

$$D(t, p * [] * E') = D(D(t, p' * [] * [x \mapsto \$, m] \cdot E), [x \mapsto B]) \qquad (lemme\ 10)$$
$$= D(t, p' * [] * [x \mapsto \$, m] \cdot E)\langle\langle D(B, ., E')/x\rangle\rangle\{0/x\} \quad (lemme\ 12)$$
$$\text{Or } D(\lambda x.t, m * B, n * E) = (\lambda x.D(t, m * [x \mapsto \$] \cdot E))D(B, n * [] * E')$$
$$\longrightarrow_\beta D(t, p' * [] * [x \mapsto \$, m] \cdot E)\langle\langle D(B, ...E)/x\rangle\rangle\{0/x\}$$

D'où $D(\kappa_1) \xrightarrow{1}_\beta D(\kappa_2)$ □

Theorème 14. $\longrightarrow_\kappa$ *effectue bien la* β*-réduction à grand pas, de tête, faible.*

Démonstration. La seule chose qu'il nous reste à prouver, c'est que la machine ne bégaie pas, *i.e.* qu'elle ne fait pas de surplace d'un point de vue de la β-réduction.

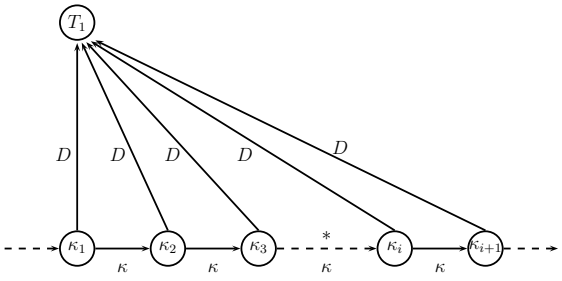


FIGURE 12 – Le risque de bégaiement

Si le terme admet une forme normale de tête (seul cas intéressant), il y a un nombre fini de pas par β-réduction donc un nombre de *grab* fini lui aussi (d'après le lemme précédent). *push* et *access* font strictement décroître un ordre lexicographique bien choisi ((*taille env courant*, *taille terme de tête*)) donc sont en

9. *Une fois encore, on triche sur les notations, afin de mieux voir ce qu'il se passe.* $D(X, E)D(Y, E')$ *tient lieu de* $\sum_{i,j} \delta(E_{i,j}) \cdot x_i y_j$ *où* $\begin{cases} \mu(X, E) = \sum_i x_i, E_i \\ \mu(Y, E_i) = \sum_j y_{i,j}, E_{i,j} \end{cases}$

nombre fini entre deux *grab*. Au final, on peut clairement ramener la chose à un arbre à branchements finis, qui est donc lui-même fini (d'après le Lemme de König). □

3.4 Normalisation

Ceci étant, la deuxième chose qui peut nous intéresser est de connaître la forme des termes sur lesquels la machine s'arrête, et notamment en terme de forme normale pour une certaine réduction. On définit pour cela $\mathcal{K}(\kappa)$ comme étant la machine obtenue par $\overset{*}{\longrightarrow}_\kappa$ avec en tête la forme normale de κ pour $\longrightarrow_\kappa$ si elle existe (auquel cas elle est bien unique d'après la propriété 6), Ω sinon. On a alors le résultat suivant :

Proposition 15. *Si la machine s'arrête, alors sa traduction est en forme normale de tête faible* fntf *(i.e. pour la réduction de tête faible). Plus formellement,*

$$\forall t, \mathcal{K}(i(t)) = \Omega \text{ ou } D(\mathcal{K}(i(t))) \text{ en fntf}$$

Démonstration. Si la machine s'arrête dans une branche, de deux choses l'une, soit le terme courant se réduit en 0, soit c'est une λ-abstraction devant une pile vide (et on ne peut faire de *grab*), le terme est donc bien en forme normale de tête faible, il ne reste qu'à le décompiler. Il en va de même pour chaque branche, de fait, on en déduit que si la machine s'arrête, tous les termes de la décompilation sont en forme normale de tête faible. □

Cependant, cette normalisation n'est guère satisfaisante, nous allons voir par la suite comment l'améliorer.

4 Discussion

4.1 Une normalisation plus forte

On aimerait retrouver un comportement normalisateur de la machine, pour une relation de réduction plus forte. On peut relativement facilement simuler une réduction de tête, en continuant l'exécution sous les λ. Pour ce faire, on agrandit un petit peu la machine, en rajoutant la liste des λ sous lesquels on travaille :

$$\kappa \equiv \sum([\lambda_1 \dots \lambda_n] * t, m * \pi * E)$$

On ajoute de plus, comme pour la décompilation, un symbole \$ au langage des termes afin de marquer les termes constants. On étend $\longrightarrow_\kappa$ à $\longrightarrow_{\bar{\kappa}}$ ainsi : on conserve les règles $push, access1$ et $access2$, en ajoutant de part et d'autres de la flèche les λ de têtes, et l'on rajoute la règle suivante :
Under-λ :

$$hd * \lambda x.t, m * \emptyset * E \longrightarrow_{\bar{\kappa}} \lambda x :: hd * t, m' * \emptyset * E' \text{ avec } \begin{cases} m' \text{ frais} \\ E'(m') = x, ([\$x], m), m \\ E'(m) = E(m) \quad \forall m \neq m' \end{cases}$$

Le $\$x$ permet de marquer que x est lié « au-dessus », et, n'étant pas pris en compte dans les règles *access*, cela garantit l'arrêt de la machine devant une telle variable. On adapte très naturellement la décompilation, en remettant tout simplement en tête les λ de côté.

On a ainsi construit une machine abstraite permettant de descendre sous les λ tout en ayant le même comportement que la version initiale dans les autres cas. On en déduit facilement le résultat suivant :

Theorème 16. $\longrightarrow_{\bar{\kappa}}$ *effectue bien la* β*-réduction à grand pas, de tête.*

De même, d'un point de vue de la normalisation, on obtient, en notant $\bar{\mathcal{K}}$ le pendant de $\mathcal{K}$:

Proposition 17. *Si la machine s'arrête, alors sa traduction est en forme normale de tête* fnt*. Plus formellement,*

$$\forall t, D(\bar{\mathcal{K}}(i(t))) = \Omega \text{ ou } D(\bar{\mathcal{K}}(i(t))) \text{ en fnt}$$

4.2 Lien avec la solvabilité

Cependant, l'un des intérêts des termes normalisables de tête en λ-calcul usuel, c'est qu'ils caractérisent les termes dits solvables. Or la solvabilité, telle qu'elle est introduite en λ-calcul avec ressources [PRDR10], induit une autre définition des formes normales. En effet, usuellement, un terme est dit solvable lorsque l'on peut, dans un contexte tête simple, le réduire en n'importe quel autre terme, ou l'identité, c'est équivalent. L'idée étant de dire que ce terme va pouvoir nous être utile.

En λ-calcul avec ressources, les choses se compliquent un petit peu, puisque les arguments linéaires doivent être impérativement utilisés, sous peine de mort par abondance. De fait, on a besoin qu'ils soient eux aussi solvables. Ce qui amène naturellement à cette autre définition :

Définition 18 ([PRDR10])**.** Un terme est dit en **forme normale externe** (*hnf* par homogénéité avec la version anglaise de la chose) s'il ne comporte plus de redex ailleurs que sous des $(.)^!$. On peut ainsi définir l'ensemble des *hnf* comme suit :

$$\begin{array}{rl} \lambda x.M \text{ est en } \textit{hnf} & \text{si } M \text{ est en } \textit{hnf} \\ xP_1 \dots P_p \text{ est en } \textit{hnf} & \text{si } p \geq 0 \\ & \text{et } \forall i \leq p, \text{ chaque ressource linéaire dans } P_i \text{ est en } \textit{hnf} \end{array}$$

On construit l'ensemble des contextes de tête simples de la manière suivante :

$$\Lambda_{(.)} : \qquad C(\cdot), D(\cdot) := (\cdot) \mid \lambda x.C(\cdot) \mid C(\cdot)P$$

Enfin, deux notions de solvabilité différents existent, on peut soit vouloir que, pris dans un contexte $C(\cdot)$, quel que soit le réduit choisi, celui-ci vaille l'identité $\mathbf{I}$ (*must-solvabilité*) ; soit que, parmi les réduits, il en existe un qui vaille l'identité (*may-solvabilité*), ce qui correspond à la définition suivante :

Définition 19. Un terme M est dit solvable s'il existe un contexte de tête $C(\cdot)$ simple et une somme de termes $\mathbb{N}$ tels que $C(M) \longrightarrow_\beta^* \mathbf{I} + \mathbb{N}$

En se conformant à ces définitions, on a le résultat suivant :

Theorème 20 ([PRDR10])**.** *Étant donné un terme avec ressources M, les propositions suivantes sont équivalentes :*

1. *M est normalisable de tête,*

2. *M est réductible en* hnf *par réduction extérieure*
3. *M est solvable*

De fait, on aurait envie d'étendre encore la machine pour s'aligner sur cette définition. En fait, c'est tout à fait possible, on procède comme pour le passage sous les λ, sauf que cette fois-ci on cherche à aller aussi dans les paquets des termes à mettre en tête. Au lieu de garder en plus les λ de tête, il faut alors conserver un contexte. La difficulté étant alors de pouvoir naviguer à l'intérieur du terme complet, par exemple dans plusieurs éléments d'un même paquet. On a alors besoin de connaître le contexte sous une structure de Zipper. Ce qui relève alors plus de la technique d'implémentation.

Cependant, moralement, si l'on en arrive à cette solution, il n'est alors pas plus compliqué de normaliser aussi les arguments linéaires. On retrouve alors la β-réduction dans son intégralité. Ce qui remet quelque peu en cause le choix de définition de *hnf*.

4.3 Implémentation, optimisation

Lors de ce stage, la machine abstraite (avec les λde tête) ainsi que les fonctions de décompilation correpondantes ont été implémentées en CamL. Diverses optimisations sont réalisables (et ont été réalisées), s'inspirant notamment de choses existant pour la machine de Krivine usuelle. En voici les principales.

La première des envies, ou tout du moins une envie très légitime, serait de détecter aussitôt que possible les termes courant droit à leur perte . Pour la famine, c'est difficilement détectable puisque, mis à part ce qui est en tête du terme de tête, on ne sait pas trop prédire ce qui va servir ou non (on peut avoir des fonctions qui jettent leurs arguments, etc...). En revanche, pour l'abondance, on peut en détecter un certain nombre à la volée. Cela nécessite de savoir à tout instant combien d'éléments pointent indirectement ou non sur un paquet, donc de faire un comptage de références, et alors, chaque fois que plus personne ne pointe sur un paquet (ce qui signifie que l'on est sûr qu'il ne servira plus jamais), on regarde si celui-ci contient ou non encore des arguments linéaires. Ce n'est pas excessivement dur, bien qu'il faille être relativement précautionneux, et en revanche, cela nous apporte beaucoup d'information sur la machine tout au long de l'exécution.

Une autre amélioration, relativement classique [Wan03] pour la machine usuelle, consiste, lors du *grab*, à aller tout de suite au bout du lien. En effet, lorsque l'on observe l'exécution de $\delta\delta$, on observe l'apparition dans l'environnement d'une chaîne $[x \mapsto x]\cdot[x \mapsto x]\cdot[x \mapsto x]\cdots[x \mapsto \delta]$. Ce qui peut aussi arriver en λ-calcul avec ressources, mais seulement dans le cas de paquets ne contenant qu'une variable. Et se rajoute alors la difficulté de ne pas perdre en cours de route les multiplicités. C'est un cas peut-être restrictif de manière générale, mais en revanche courant si l'on s'intéresse à des termes issus du λ-calcul usuel. En y regardant de plus près, en considérant l'environnement comme une sorte d'arbre (avec pour feuilles des « vraies »ressources (*i.e.* des termes clos), cela nous ramène à un problème de flots facile à traiter.

Enfin, en considérant toujours l'exécution d'un terme d'une somme comme un choix d'exploration d'un arbre, on peut aussi être amené à réfléchir à un système de *garbage collecting*, afin de minimiser l'espace occupé par ledit arbre

en mémoire. On se rapproche néanmoins d'un travail d'implémentation pur, plus très en lien théorique avec la machine.

4.4 Une autre version de la machine

La façon de séparer un paquet lors de la substitution à une application (cf. lemme 11) peut donner l'idée de travailler sans partage de ressources, en les attribuant de façon définitive lors du *push*, en travaillant avec la grammaire de la machine de Krivine usuelle.

De fait, cela simplifie amplement la décompilation [10], puisque l'on peut se passer du travail avec continuation, et fonctionner de façon purement récursive [11]

On calque donc la séparation d'environnement sur ce qui se passe lors d'une substitution :

$$(tU)\langle\langle B/x\rangle\rangle = \sum_{A \lhd B} (t\langle\langle A/x\rangle\rangle)(U\langle\langle \bar{A}^B/x\rangle\rangle)$$

$$\Updownarrow$$

$$split = \begin{cases} \emptyset \mapsto \emptyset, \emptyset & \\ [x \mapsto B, \rho'] \cdot \rho \mapsto & \bigcup_{\rho_1} \bigcup_{\rho'_1} \bigcup_{A \lhd B} [x \mapsto A, \rho'_1] \cdot \rho_1, [x \mapsto \bar{A}^B, \rho'_2] \cdot \rho_2, \\ & \text{où } split(\rho) = \bigcup_{\rho_1} \rho_1, \rho_2 \text{ et } split(\rho') = \bigcup_{\rho'_1} \rho'_1, \rho'_2 \end{cases}$$

Moralement, pour couper un environnement, on le parcours récursivement, et à chaque lien avec un paquet, on coupe le paquet en deux de façon idoine (en dupliquant les ressources non-linéaires). On peut alors construire les pas d'exécution, en coupant l'environnement au *push*, et en vérifiant la non-abondance lors de l'*access* (puisque l'on jette définitivement le paquet).

Push :	$t\ B, \rho * \pi$	$\longrightarrow_\kappa$	$\sum_{(\rho_1, \rho_2) \in split(\rho)} t, \rho_1 * (B, \rho_2) :: \pi$
Access1 :	$x, [y \mapsto C] \cdot \rho' * \pi$	$\longrightarrow_\kappa$	$\delta(C) \cdot x, \rho' * \pi$
Access2 :	$x, [x \mapsto B, \rho'] \cdot \rho'' * \pi$	$\longrightarrow_\kappa$	$\delta(\rho'') \cdot \sum_{b^{(!)} \in B} \delta(B \backslash \{b^{(!)}\}) b, \rho' * \pi$
Grab :	$\lambda x.t, \rho * C :: \pi$	$\longrightarrow_\kappa$	$t, [x \mapsto C] \cdot \rho * \pi$

FIGURE 13 – Pas d'exécution, version sans partage

10. *La conception de fonctions idoines est gracieusement laissée au lecteur passionné, qui constatera la véracité du propos*

11. *Il sera de bon goût pour le susdit lecteur de s'appuyer sur la fonction* split*, afin de pouvoir écrire des choses comme* $D(tu, \rho) = \sum D(t, \rho_1) D(u, \rho_2)$.

5 Conclusion

Je pense a posteriori que l'objectif principal a été globalement rempli, puisque l'on dispose bien d'une machine abstraite pour le λ-calcul avec ressources. On en a en fait même deux, bien distinctes, dont il serait intéressant de comparer les avantages respectifs.

Pour ce qui est de mon travail, il a été découpé en grandes phases, ayant toutes une démarche propre, ce qui m'a permis, je pense, de varier les plaisirs, et de découvrir plusieurs aspects du domaine de la recherche. Dans un premier temps, je me suis focalisé sur la construction d'une machine correcte, ce qui passe par des tentatives systématiques de trouver la faille dans ce que l'on vient de concevoir, et une implémentation au fur et à mesure. Ensuite, m'étant convaincu de la validité de la machine dans sa dernière version, je me suis lancé dans la preuve de sa correction, à la recherche d'une technique adaptée. Puis j'ai passé un peu de temps à améliorer l'implémentation, et à voir la signification théorique des différents "patchs". Enfin, j'ai essayé de faire le lien avec d'autres sujet de recherche, dont un relativement récent, la solvabilité en λ-calcul avec ressources.

De fait, ce stage aura été pour moi une expérience aussi riche que plaisante, ce dont je ne saurais trop remercier Lionel et Emmanuel.

Références

[Bou93] Gérard Boudol. The lambda-calculus with multiplicities (abstract). In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 1993.

[ER03] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309 :2003, 2003.

[ER05] Thomas Ehrhard and Laurent Regnier. Böhm trees, krivine machine and the taylor expansion of ordinary lambda-terms. available at http ://iml.univ-mrs.fr/ $\sim$ ehrhard/pub/bkt.ps.gz, 2005.

[Kri90] Jean-Louis Krivine. *Lambda-Calcul : Types et Modèles*. Études et Recherches en Informatique. Masson, 1990.

[Kri04] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In *Higher Order and Symbolic Computation*, 2004.

[Lan07] Frédéric Lang. Explaining the lazy krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation*, 20 :257–270, 2007. 10.1007/s10990-007-9013-1.

[PRDR10] Michele Pagani and Simona Ronchi Della Rocca. Solvability in Resource Lambda-Calculus. In Luke Ong, editor, *FOSSACS*, volume 6014 of *Lecture Notes in Comput. Sci.*, pages 358–373, 2010.

[PT09] Michele Pagani and Paolo Tranquilli. Parallel Reduction in Resource Lambda-Calculus. In Zhenjiang Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Comput. Sci.*, pages 226–242, 2009.

[Wan03] M Wand. On the correctness of the krivine machine. In *In Danvy [64]. In preparation*, 2003.