# Compilation des règles de récriture vers la ZAM

June 29, 2005

## 1 Le $\lambda$-calcul avec des règles de récriture

### 1.1 Les termes et les règles

Nous considérons le langage suivant, composé de règles de récriture $r$ et de termes $t$ :

$$\begin{array}{rcl} t & := & c \mid x \mid \lambda x.t \mid (t\ t) \\ r & := & c\ t_1\ ...\ t_n \rightarrow t_{n+1} \end{array}$$

Les constantes sont dénotées par $c$, lors d'une règle de récriture, nous devons toujours avoir une constante de tête. L'idée est de pouvoir définir une fonction par une ou plusieures règles de récriture.

Les variables sont dénotées $x$, etc, etc ...

Les règles de récriture sont des règles qui font appel à du filtrage du premier ordre. En particulier, on ne peut pas avoir de $\lambda$ dans le membre gauche. De plus, nous devons vérifier la condition suivante (habituelle pour les règles de récriture) : les variables libres du terme $t_{n+1}$ sont incluses dans celles des termes $t_1, ..., t_n$. Ce qui peut se résumer par :

$$FV(c\ t_1\ ...\ t_n) \quad \supseteq \quad FV(t_{n+1})$$

Exemple de règles de récriture :

$$\begin{array}{rcl} +0 & \rightarrow & id \\ +0(Sx) & \rightarrow & S(+0x) \end{array}$$

où $x$ est une variable.

### 1.2 Les règles de réduction

Elles sont au nombre de deux : nous avons la classique $\beta$-réduction, et, puisque nous avons des règles de récriture, nous allons introduire une règle de réduction pour celles-ci.

Voici les règles :

$$\begin{array}{rcl} \lambda x.t\ t' & \rightarrow & t_{[x \leftarrow t']}\ (\beta) \\ c\ t'_1\ ...\ t'_n & \rightarrow & t_{n+1}\sigma\ \text{(rewriting)} \end{array}$$

La règle (*rewriting*) s'applique sous les conditions suivantes :

1. Qu'il existe une règle de récriture $c\ t_1\ ...\ t_n \rightarrow t_{n+1}$, qui ait été explicitement définie auparavant (en général, on les regroupe dans un ensemble appelé $\mathcal{R}$).
2. Que $c\ t'_1\ ...\ t'_n$ soit unifiable avec $c\ t_1\ ...\ t_n$, et que $\sigma$ représente la substitution associée. Comme nous sommes en first-ordre matching, s'il y en a une, elle est unique, et il est relativement facile de la trouver.

Dans toute la suite, nous ne nous intéressons pas à la terminaison de l'algorithme de réduction forte (ou à n'importe quel algorithme de réduction d'ailleurs). Ce n'est pas notre travail. Nous supposerons simplement la normalisation forte, c'est à dire que n'importe quelle stratégie de réduction *termine*. À la charge d'autres personnes de le démontrer (par exemple, avec le critère HORPO, ou bien la thèse de Frédérique Blanqui).

## 1.3 Passage aux indices de De Bruijn

Marre des variables et de l'$\alpha$-conversion. Nous passons aux indices de De Bruijn, en plus, comme le dit très bien Benjamin Grégoire, c'est comme ca que c'est implanté dans Coq (et dans la ZAM), donc on ne va pas se priver, puisqu'on veut la réutiliser.

Le travail de passage en indices de De Bruijn pour les $\lambda$-termes a déjà été effectué. Mais nous devons encore le faire pour les règles de récriture. Pour cela, nous devons faire apparaître des lieurs quelque part. En effet, pour les $\lambda$ termes, le lieur est $\lambda$, par exemple :

$$\lambda.f0\ c \quad \rightarrow \quad fc$$

Intuitivement, les lieurs sont les variables libres du membre droit d'une règle de récriture. Par exemple, dans la règle

$$+(Z)x \rightarrow x$$

le premier $x$ est lieur et le $x$ à droite est lié. ($Z$ représente la constante 0, que nous ne pouvons plus nommer ainsi à cause des indices de De Bruijn).

L'idée est donc de distinguer les occurences lieuses des occurences liées, puisqu'elles n'ont pas du tout la même fonction dans la règle de récriture. L'occurence lieuse sera remplacée par ?, qui remplit le rôle d'un $\lambda$, et l'occurence liée sera remplacée par un indice de De Bruijn. Ainsi, les règles pour l'addition auront la forme :

$$\begin{array}{rcl} +(Z)? & \rightarrow & 0 \\ +S(?)? & \rightarrow & S(+10) \end{array}$$

Des “trous” devraient apparaissent dans le membre gauche, représentant des instances lieuses, et des indices de De Bruijn à droite, qui représentent dse instances liées.

Une difficulté supplémentaire vient des ˚‘egles de récriture non linéaires, tels que par exemple

$$fxx \rightarrow gx$$

En fait, dans cette règle, dans le membre gauche $fxx$, le premier $x$ doit être considéré comme liant, et le deuxième comme “matchant”, ou bien, d’une certaine manière liée. Ainsi, il sera un indice de De Bruijn, se référant au premier $x$, qui lui, sera un “trou”, ?.

Lors de l’application d’une règle de récriture, on mettra en mémoire le premier argument de $f$, et on comparera le deuxième avec celui en mémoire. Ceci introduit une dissymétrie qui n’est pas présente dans la définition originelle de la règle de récriture. Elle correpond à l’application “pratique” de cette définition, ce qui montre que les indices de De Bruijn sont plus proches de l’implémentation que les définitions avec dse noms de variables.

Le langage auquel nous arrivons est présenté dans la figure suivante. Notons l’introduction d’une nouvelle classe syntaxique, les patterns $p$, qui ne peuvent exister par elles-mêmes dans le calcul.

$$\begin{array}{rcl}
t & := & c \mid n \mid \lambda.t \ \mid (t\ t) \\
p & := & c \mid n \mid ? \mid (p\ p) \\
r & := & c\ p_1\ ...\ p_n \rightarrow t_{n+1}
\end{array}$$

Nous avons la condition suivante dans les règles de récriture : elles doivent être closes, c’est à dire que tous les indices de De Bruijn présents dans $t_{n+1}, p_1, ..., p_n$ doivent se référer à une instance lieuses ? déjàprésente auparavant.

Il faudrait donc définir ce qu’est une variable libre et une variable liée dans les règles de récriture. Au lieu de cela, voici une méthode très simple. Nous allons définir une translation des règles vers les lambda-termes non typés, que nous appelleront $T$.

**Definition 1 (Clôture d’une règle de récriture)**

$$\begin{array}{rcl}
T[c\ p_1\ ...\ p_n \rightarrow t_{n+1}] & := & T[c]\ T[p_1]\ ...\ T[p_n]T[t_{n+1}] \\
T[c] & := & c \\
T[(p_1 p_2)] & := & T[p_1]T[p_2] \\
T[?] & := & \lambda. \\
T[n] & := & n
\end{array}$$

*Une règle de récriture $r$ est close ssi $T(r)$ est un terme clos.*

Remarquons que par exemple $c(?(?c))? \to 3$ est transformée en $\lambda.\ \lambda.c\lambda.3$ , et non pas en $(\lambda.\ \lambda.c\ )\lambda.\ 3$ et qu'elle est donc valide en tant que règle de récriture. En effet, la parenthèse n'arrête pas le scope de la variable lieuse. D'ailleurs, sinon cela conduirait à une translation produisant des termes mal-formés, comme dans le cas de $c(??)?$.

Si on veut passer par le chemin normal, il vaut mieux passer pa rle calcul symbolique, qui donne une représentation explicite du nombre de variables liées dans un pattern.

Par exemple les deux premières règles est une règle valide et les deux dernières, non.

$$\begin{array}{lcl} f?? & \to & (\lambda.3\ ) \\ f?0 & \to & g \\ f?? & \to & g3 \\ f0? & \to & g0 \end{array}$$

Si l'on a une règle de récriture syntaxiquement bien formée, on peut la transformer de manière automatique en règle de récriture bien formée pour ce calcul. Chose à définir précisément, voir section 3..

Intuitivement, la première règle correspond à la règle (sans De Bruijn) suivante :

$$fxy \to \lambda z.x$$

Ils nous faut maintenant donner immédiatement les règles de réduction associées.

Nous avons toujours la $\beta$-réduction, et la récriture :

$$\begin{array}{rcl} \lambda.t\ t' & \to & t[0 \leftarrow t'] \\ c\ t'_1\ ...\ t'_n & \to & t_{n+1}[0 \leftarrow \overrightarrow{y}] \end{array}$$

Dans la règle de récriture (rewriting), nous avons maintenant la substitution parallèle $[0 \leftarrow \overrightarrow{y}]$, qui représente le résultat (s'il y en a un) du matching entre $c\ t'_1\ ...\ t'_n$ et $c\ p_1\ ...\ p_n$.

Matching qui reste encore à définir. Voire la section 2.

Remarquons par contre que réduire une RR peut faire intervenir des $\beta$-redexes qui n'y étaient pas avant ... ceci est à prendre en considération, voir sections suivantes (et la section de normalisation).

# 2 Matching en indices de De Bruijn

Nous définissons dans cette section une fonction $\Phi$, qui prend un problème de pattern matching en argument $p = t$ (pattern/terme). Elle prend un autre argument, qui est la substitution partielle déjà définie (nous la noterons en indice). En effet, elle peut prendre un problème déjà partiellement résolu, et dans ce cas, on peut trouver des variables libres dans $p$, c.à.d des indices de De Bruijn non liés. Ils devront être liés dans la substitution (du moins, si le pattern est bien formé).

$\Phi$ renvoie la substitution associée si ça marche, et une exception sinon.

Voici sa définition (rappelons que nous faisons du matching du premier ordre, donc pas de $\lambda$ dans les patterns. En particulier, on ne peut pas "rentrer" sous un $\lambda$ dans le matching, car matcher $P$ et $\lambda.t$ donnera toujours une erreur, sauf dans le cas de variables libres ? ou liées $n$ pour le pattern) :

$$
\begin{array}{rcl}
\Phi(c = c)_{[0 \leftarrow \overrightarrow{x}]} & \triangleright & [0 \leftarrow \overrightarrow{x}] \\
\Phi(c = t)_{[0 \leftarrow \overrightarrow{x}]} & \triangleright & fail \;\; (1) \\
\Phi(? = t)_{[0 \leftarrow \overrightarrow{x}]} & \triangleright & [0 \leftarrow t.\overrightarrow{x}] \\
\Phi(m = t)_{[0 \leftarrow \overrightarrow{x}]} & \triangleright & [0 \leftarrow \overrightarrow{x}] \text{ if } t = x_m (2) \\
\Phi(m = t)_{[0 \leftarrow \overrightarrow{x}]} & \triangleright & fail \text{ if } t \neq x_m \\
\Phi(t_1 t_2 = p_1 p_2)_{[0 \leftarrow s]} & \triangleright & \Phi(t_2 = p_2)_{\Phi(t_1 = p_1)_{[0 \leftarrow s]}} (3)
\end{array}
$$

Quelques remarques : (1) si $t$ n'est pas la constante $c$, évidemment.

(2) est caractéristique du filtrage du premier ordre ... sinon on ferait des tests de conversion entre $x_m$ et $t$.

(3) C'est le coeur du problème. On matche $t_1$ et si ça marche, on matche $t_2$ sous la substitution associée à $t_1$.

Reste à savoir si on ne laisse rien passer avec ce genre de filtrage ?

En tous cas, ca laisse un peu à désirer, car on n'aura pas le même résultat en fonction de l'ordre d'évaluation. La RR :

$$f?0 \rightarrow g$$

donne :$f(\lambda.0\ c)c$ ne se réduit pas, ou bien se réduit, c'est selon la $\beta$ réduction ... mais ce n'est pas MON Pb ? Car ici, on n'est censé ne filtrer que des valeurs, et ce qu'on a en premier argument n'est pas une valeur.

De toutes façons, on obtient au pire des problèmes de confluence, qui n'entrent pas dans notre scope.

Par contre, on peut avoir :

$$f?? \rightarrow 10$$

et $f(\lambda.0\ )c$. après RR-réduction, donne $\lambda.0\ c$, qui sera à $\beta$-réduire. Donc il faut faire gaffe à l'algorithme de normalisation ...

# 3 Transformer des règles en indices de De Bruijn

Ça, ce n'est pas trop difficile.

Donc, on définit une transformation automatique, modulo un ensemble$\{\overrightarrow{x}\}$ de noms de variables liées (l'ordre a une importance). Moralement, à chaque indice dans le vecteur est associé un nom de variable (qui a déjà été rencontré dans le passé). On récupère à la sortie un couple composé du terme transformé et du vecteur $\{\overrightarrow{x}\}$ :

$$
\begin{array}{rcl}
y^{\overrightarrow{x}} & \rhd & (i, \overrightarrow{x}) \text{ si } y = \overrightarrow{x}[i] \\
y^{\overrightarrow{x}} & \rhd & (?, y.\overrightarrow{x}) \text{ sinon} \\
c^{\overrightarrow{x}} & \rhd & (c, \overrightarrow{x}) \\
(p_1 p_2)^{\overrightarrow{x}} & \rhd & (p'_1 p''_2, \overrightarrow{x''}) \\
l \to r & \rhd & fst(l^{\overrightarrow{0}}) \to fst(r^{snd(l^{\overrightarrow{0}})})
\end{array}
$$

(1) Ici, on appelle $p_1^{\overrightarrow{x}} = (p'_1, \overrightarrow{x'})$, et $p_2^{\overrightarrow{x'}} = (p''_2, \overrightarrow{x''})$.

NB: les membres gauches sont tous en forme $\beta$-normales, puisqu'aucun $\lambda$ n'y apparaît. Par contre, on pourrait vouloir imposer le fait qu'ils soient en forme RR-normale, ce qui n'est pas tout le temps le cas :

$$f(+0x) \quad \to \quad g$$

En effet, on aurait des problèmes de confluence avec cette règle. Mais encore une fois, ce n'est pas notre problème, et tout ce que nous avons défini marche très bien, même sans cette restriction.

Par contre, on ne peut pas esperer fixer un membre droit en NF (que ça soit pour la règle de récriture ou pour la $\beta$-réduction), cf exemple section précédente..

# 4 Le langage symbolique

Un bon morceau du boulot. On va définir un langage symbolique et une réduction associée qui simule la réduction $\beta$ et rewrite du $\lambda$-calcul.

L'idée est de mettre la RR en indice de la constante de tête. Le Pb, c'est que ça peut devenir circulaire. Par exemple $+xS(y) \to S(+xy)$. On a encore un signe plus, qui devrait-être lui-même expansé. Pour éviter cela, on va définir des constantes tildées, et des réductions appropriées.

Les termes du calcul symbolique sont organisés en plusieurs catégories syntaxiques. $t$ représente un terme. $k$, un accumulateur. $v$ une valeur (c'est un terme dont on ne peut attendre de réduction "de tête"). Et bien sûr, les règles de récriture $r$ et les patterns $p$. La nouveauté se situe ici. Les patterns ont pour indice le nombre de variables liées. Les règles de récriture ont une catégorie en plus : les règles $\widetilde{c}$ qui représentent la fin des RR (une constante qui ne peut plus se récrire). Et enfin, la catégorie des valeurs accepte une valeur en plus : des symboles définis par des RR partiellement appliqués (et donc, qu'on ne peut pas réduire par les RR pour le moment) :

$$
\begin{array}{rcl}
t & := & n \mid (t\ t) \mid c \mid v \\
k & := & \widetilde{c} \mid k\ v \mid n \\
v & := & \lambda.t \ \mid [k] \mid \ \parallel \overline{c_r}\ v_1\ ...\ v_m \parallel \\
r & := & \ if\ (p_1, ..., p_n)\ then\ t\ else\ r \ \mid \widetilde{c} \\
p & := & c_{/0} \mid \square_{/1} \mid n_{/0} \mid (p_{/n} p_{/m})_{/m+n}
\end{array}
$$

Une condition tout de même sur les règles de récriture: le nombre $n$ de patterns $(p_1, ..., p_n)$ est supérieur ou égal au nombre de patterns de la règle $r$ suivante, avec comme convention que si la règle est $\widetilde{c}$, alors elle a un nombre de pattern infini.

La réduction faible :

$$
\begin{array}{rcll}
[k]v & \mapsto & [kv] & (1) \\
\lambda.t\ \ v & \mapsto & t[0 \leftarrow v] & (2) \\
\parallel c_{\ if\ (p'_1,...p'_n)\ then\ t''\ else\ r_1}\ v_1\ ...\ v_m \parallel v_{m+1} & \mapsto & & (3) \\
 & & \parallel c_{\ if\ (p'_1,...p'_n)\ then\ t''\ else\ n_1, r_1}\ v_1\ ...\ v_{m+1} \parallel & \\
\parallel \overline{c}_{\ if\ (p'_1,...p'_n)\ then\ t''\ else\ r_1}\ v_1 ... v_{n-1} \parallel v_n & \mapsto & t''[0 \leftarrow \overrightarrow{x}] & (4) \\
\parallel \overline{c}_{\ if\ (p'_1,...p'_n)\ then\ t_2\ else\ r_1}\ v_1 ... v_{n-1} \parallel v_n & \mapsto & \parallel \overline{c}_{r_1} v_1 ... v_{n-1} \parallel v_n & (5) \\
\parallel \overline{c}_{\tilde{c}}\, v_1 ... v_{n-1} \parallel v_n & \mapsto & [\tilde{c} v_1 ... v_{n-1} v_n] & (6) \\
c & \mapsto & \parallel \overline{c_r} \parallel & (7) \\
\Gamma(t) & \mapsto & \Gamma(t') \text{ if } t \mapsto t' & (8)
\end{array}
$$

Conditions: Dans la troisième équation, il faut que $m + 1 < n$, dans la quatrième la condition est si le matching matche et dans la cinquième s'il ne marche pas, avec $[0 \leftarrow \overrightarrow{x}] = \Phi(cp'_1...p'_n = cv_1...v_n)$.

Enfin, la sixième équation n'a pas de condition particulière et s'applique tout le temps.

La septième équation ne s'applique qu'à des constantes qui possèdent une règle de récriture.

# 5 La langage symbolique étendu

Nous remplaçons simplement la construction $\ if\ c\ then\ a\ else\ b\ $ par une construction de matching. Voici la définition des termes de type $r$.

$$r := match(n) with \quad p_{11}, ..., p_{1n} \quad \mapsto t_1$$

$$
\begin{array}{lll}
| & p_{21}, ..., p_{2n} & \mapsto t_2 \\
| & \vdots & \\
| & p_{k1}, ..., p_{kn} & \mapsto t_3 \\
| & default & \mapsto r \\
 & OU\ BIEN & \\
\widetilde{c} & &
\end{array}
$$

Tout le reste de la syntaxe est identique. Reste toujours la condition que le nombre $(n)$ de patterns doit augmenter. Ici, nous avons fixé un nombre de patterns constant dans la construction de match ... cette condition peut être relaxée en fait, mais on doit toujours avoir, du haut vers le bas, une augmentation du nombre de patterns.

En ce qui concerne la réduction faible, on change les règles de réduction numéro $3, 4$ et $5$, qui deviennent :

$$
\begin{aligned}
\parallel c_{match(n)...} v_1 ... v_m \parallel v_{m+1} &\mapsto \\
&\parallel c_{match(n)...}\ v_1 ... v_{m+1} \parallel \qquad (9) \\
\parallel \overline{c}_{match(n) with p_{11}...p_{1n} \mapsto t_1 | ... \mapsto t_k | default \mapsto r} v_1 ... v_{n-1} \parallel v_n &\mapsto t_l[0 \leftarrow \overrightarrow{x}] \qquad (10) \\
\parallel c_{match(n) with p_{11}...p_{1n} \mapsto t_1 | ... \mapsto t_k | default \mapsto r} v_1 ... v_{n-1} \parallel v_n &\mapsto \parallel \overline{c}_r v_1 ... v_n \parallel v_n \qquad (11)
\end{aligned}
$$

Cette fois-ci, la condition valable pour que la règle du milieu s'applique est que $[0 \leftarrow \overrightarrow{x}] = \Phi(cp_{l1}...p_{ln} = cv_1...v_n)$ soit bien définie. Et la règle du bas s'applique si pour tout $l \leq k$, on obtient une erreur au matching ci-dessus.

# 6 Compilation des règles de récriture

Étant donné un ensemble de règles de récriture se rapportant à une constante de tête $c$, nous construisons le terme symbolique $r_c$ associé, de telle manière que la réduction symbolique simule exactement la récriture.

Dans sa forme générale, l'ensemble de règles de récriture a la forme suivante :

$$
\begin{array}{rcl}
c\ p_{1,1}...p_{1,n_1} & \rightarrow & t_1 \\
& \vdots & \\
c\ p_{m,1}...p_{m,n_m} & \rightarrow & t_m
\end{array}
$$

Nous commençons par les classer par ordre d'arité croissante (ce que nous supposons être fait dans la définition ci-dessus).

On va définir la règle $r_c$ associée à $c$ de manière récursive, sur le *nombre d'arités différentes* des $m$ règles de récriture présentes.

Supposons que les règles soient toutes d'arité $n$, ($n_1 = ... = n_m = n$). Nous les regroupons toutes dans la même construction *match*, nous définissons "$r_c$" le terme symbolique associé à $c$ de la façon suivante :

$$\begin{array}{rcl} r_c := match(n)with & p_{1,1}, ..., p_{1,n_1} & \mapsto t_1 \\ | & p_{2,1}, ..., p_{2,n_2} & \mapsto t_2 \\ | & \vdots & \\ | & p_{m,1}, ..., p_{m,n_m} & \mapsto t_m \\ | & default & \mapsto \widetilde{c} \end{array}$$

S'il y a au moins deux arités différentes, alors soit $n$ l'arité la plus petite. Les règles de récriture ont la forme :

$$\begin{array}{rcl} cp_{1,1}...p_{1,n_1} & \rightarrow & t_1 \\ & \vdots & \\ cp_{k,1}...p_{k,n} & \rightarrow & t_k \\ \hline cp_{k+1,1}...p_{k+1,n} & \rightarrow & t_{k+1} \\ & \vdots & \\ cp_{m,1}...p_{m,n_m} & \rightarrow & t_m \end{array}$$

Considérons donc les règles $k+1$ à $m$. Nous pouvons, par récurrence, leur associer $r'_c$. Eh bien, nous le faisons, et nous définissons $r_c$ :

$$\begin{array}{rcl} match(n)with & p_{1,1}, ..., p_{1,n_1} & \mapsto t_1 \\ | & p_{2,1}, ..., p_{2,n_2} & \mapsto t_2 \\ | & \vdots & \\ | & p_{k,1}, ..., p_{k,n_k} & \mapsto t_k \\ | & default & \mapsto r_{c'} \end{array}$$

C'est tout pour la compilation des règles vers le calcul symbolique. Remarquons que c'est la définition de la réduction faible qui permet de ne pas dérouler à l'infini l'annotation sous les constantes.

*Remarque.* Important. Pour plus de tranquillité, on peut décider d'associer à chaque nom de constante n'étant pas définie par des règles de récriture UNE unique règle $\widetilde{c}$. De manière à harmoniser le traitement par la machine ensuite.

# 7 La normalisation forte

Reste à voir ... avec ces problèmes de règle de récriture qui peuvent introduire des $\beta$-redexes, et inversement !

Définissons les fonctions de normalisation et de relecture plus tard.

# 8 compilation du langage symbolique vers la machine abstraite ZAM

## 8.1 Remarques préliminaires

C'est le point le plus expérimental de notre note.

Dans la machine, à chaque fois que l'on rencontre la constante $c$, on la remplace par sa règle de récriture. (En fait, on pourrait attendre qu'elle ait le bon nombre d'argument, comme dans la réduction du calcul symbolique). Donc, il faut compiler une fois les RR, en faire un bloc, que l'on recopiera à runtime.

La règle représente le contenu calculatoire de la constante $c_r$. C'est pour cela qu'on doit faire comme ca. Le nom de la constante de tête ne nous sert strictement a rien (dans la machine).

Étant donné la structure du langage symbolique, il ne faut pas oublier que les règles de récriture filtrent absolument tout ! En effet, dans le pire des cas, on tombera dans le cas defaut ....

Dernièrement, nous nous intéressons ici à la génération du CODE lié à la règle de récriture. Ceci, on le mettra dans un bloc, qui devrait être appelé lors de la lecture de la constante.

D'autre part, où est-il mieux de mettre des GRAB ? Devant la compilation des match ? Ou bien lors de la compilation de la constante ? De manière à ne pas vouloir expanser à tout prix une constante qui n'a pas encore reçu le bon nombre d'arguments ... Un match (n) doit se transformer en GRAB(n) ou quelque chose d'approchant.

L'idée est de compiler à la suite tous les éléments du match donc, grossomodo, chacune des règles de récriture. Ainsi, si un matching foire, on passe au matching suivant. Et s'il réussit, on passe à la compilée du terme courant.

Le problème technique vient du fait que pour faire le matching, nous sommes obligés d'utiliser la pile, et d'empiler plein de valeurs, termes, etc, etc, qu'il faut mettoyer ensuite. Ce n'est point chose aisée.

C'est pour cela qu'il faut tenir en mémoire (lors de la définition de la compilation) beaucoup de choses à la fois.

## 8.2 Définition de la fonction de compilation

Donc, pour la compilation des patterns, j'utilise l'environnement de compilation suivant :

1. `Lbl` : le label ou sauter en cas d'echec du matching.

2. `Cont` : la continuation en cas de succes du matching. Il peut être de deux types.

   a Un terme $T$, qui représente un des termes $t_1, ..., t_m$ dans la construction *match*.

   b une "continuation de matching", c'est à dire que nous avons encore un autre argument à pattern-checker. La continuation sera représentée ici par un triplet $(P, m', \texttt{Cont})$, où $P$ est le nouveau pattern, `Cont` est elle-même la continuation du pattern $P$ et $m'$ est l'indice d'accès (dans la pile) à la valeur à pettern-matcher.

3. $\rho$ : l'acces aux variables deja instanciees par le pattern matching. $\rho$ sert à repérer les instances introduites par ?, dont on aura besoin ensuite. $\rho$ associe à tout indice de De Bruijn valable une position dans la pile (à partir du sommet, 0 pour le dessus de la pile).

4. $n$ : le nombre de variables deja instanciées. C'est à dire l'indice de De Bruijn maximum qui est valide pour $\rho$, ou, dit autrement, la taille de $\rho$.

5. $m$ : le nombre d'éléments que l'on a empilé sur la pile, depuis le début du pattern-matching Il faut nettoyer la pile dans le cas d'un échec du matching (et dans le cas d'un succès aussi).

   En effet, faire du pattern-matching dans la machine requiert 'enormément d'empilage de trucs (il faut bien décomposer les termes). Dès qu'on a terminé le pattern-matching (c'est à dire, quand la continuation est un terme), il faut penser à nettoyer la pile.

6. `arg` : le nombre d'arguments originel (ou le nombre de patterns originels). On a besoin de ça pour la bonne raison que si le pattern matching foire, il ne faut pas les nettoyer, et s'il réussit, il faut les nettoyer. Il faut donc garder cette information quelque part, et elle n'apparaît nulle par dans le code.

Tout ceci fait beaucoup de variables à maintenir pour la compilation. J'espère que ca deviendra plus clair par la suite.

Et donc, allons-y pour la compilation :

- Le début : la compilation d'une constante affectée d'une règle de récriture. On cherche le code associé à cette constante. Dans la pratique, on devrait, pour chaque constante faire un bloc contenant le code de la RR, et faire pointer $[\![c]\!]$ dessus. On ne le dépliera qu'en cas de besoin (c'est à dire suffisamment d'arguments).

$$[\![c_R]\!] \quad := \quad MAKEBOLCK(c, {}^{0,0,0}[\![R]\!]^{\emptyset}_{\emptyset,\emptyset});$$

On fait un bloc, de nom $c$, et un pointeur de code qui pointe sur le code compile de la constante. Quand on tombe sur un bloc fabriqué ainsi dans le code, il suffit de remplacer le bloc par le code compilé (mais seulement à ce moment là, et pas avant, pour éviter la circularité).

Pas de Label d'échec, ni de Continuation en cas de succès. C'est normal, une RR est censée filter tout. De plus, pour l'instant, on ignore s'il y a des valeurs sur la pile, et on n'a rien rajouté dessus, ce qui explique les 0.

- La compilation d'une constante tildée. C'est le cas le plus simple. Le cas final d'une RR : aucune d'entre elles ne s'applique. On crée donc un bloc, de nom la constante, et qui empile ses arguments (en principe, il y en a sur la pile, puisqu'on vient d'essayer de faire du pattern matching avec). Ensuite, ça sera la tâche de l'accu de manger les valeurs qui sont sur le sommet de la pile, tant qu'il peut.

$${}^{n,m,\mathtt{arg}}[\![\tilde{c}]\!]^{\rho}_{\mathtt{Lbl},\mathtt{Cont}} \quad := \quad MAKEACCU"c";$$

- Compilation d'un match :

$$\begin{aligned} {}^{n,m,\mathtt{arg}}[\![match(l)with p_{1,1}...p_{1,l} \mapsto t_1|...|default \mapsto t_k]\!]^{\rho}_{\mathtt{Lbl},\mathtt{Cont}} \quad &:= \\ GRAB(l - \mathtt{arg}); {}^{0,0,l}[\![p_{1,1}..., p_{1,l} \mapsto t_1|...|default \mapsto t_k]\!]^{\emptyset}_{\emptyset,\emptyset}; \\ \vdots \end{aligned}$$

Ici, on ne fait rien que déstructurer le matching. Ça nous permet de rajouter des `GRAB`(l - `arg`) devant, ce qui n'est pas un mal. On n'attend plus que $l - \mathtt{arg}$ arguments, car on sait qu'il y en a déjà `arg`arguments sur la pile ? J'avoue que je ne suis pas très au point sur ce genre de choses.

Remarquons qu'on remet (presque) tout à 0. Ce qui est normal, car al construction *match* filtre tout (grâce au cas default).

Attnetion cependant : doit on faire `GRAB l` ou bien `GRAB (l - arg)` ?? Cela reste à voir, et il faut tenir quelque part en mémoire le fait qu'il y ait déjà des arguments sur la pile (ou alors les remettre dans un bloc accumulateur ?)

Notons bien qu'il faut qu'on soit sûrs d'avoir $l$ valeurs sur le dessus de la pile. Qui seront les arguments du pattern-matching ensuite. Et nous avons la compilation avec $l$ comme argument.

- Suite du précédent cas. Compilation d'une suite de cas de matching.

  Notons que $\rho$ doit être vide. De toutes façons, il ne sert pas ici. De plus, la Continuation ne sert pas non plus, ni le label d'echec, ni même toutes les autres variables en fait, mis à part `arg`..

  On voit pour la première fois $t_1$ dans la continuation. Pour l'instant, il n'est pas compilé (ca ne viendra qu'à la fin du matching, si on a besoin). Je pense qu'on pourrrait le compiler sans trop de soucis.

  On voit aussi pour la première fois un label qui a du sens dans le cas d'échec. Effectivement, si le pattern matching échoue, alors il faut tenter les autres RR, ce qui est mis dans $\mathtt{Lbl}(reste)$.

$$\begin{array}{rcl} {}^{n,m,\mathtt{arg}}[\![p_{1,1}...,p_{1,l} \mapsto t_1|...|default \mapsto r]\!]^{\rho}_{\mathtt{Lbl},\mathtt{Cont}} & := & {}^{0,0,\mathtt{arg}}[\![p_{1,1}...p_{1,n}]\!]^{\emptyset}_{\mathtt{Lbl}(reste),t_1} \\ \mathtt{Lbl}(reste) & : & \\ {}^{n,m,\mathtt{arg}}[\![p_{2,1},...,p_{2,n} \mapsto t_2|...|default \mapsto t_k]\!]^{\rho}_{\mathtt{Lbl},\mathtt{Cont}} & & \end{array}$$

  La deuxième possibilité est qu'on soit déjà sur le cas $default$ :

$${}^{n,m,\mathtt{arg}}[\![default \mapsto r]\!]^{\rho}_{\mathtt{Lbl},\mathtt{Cont}} \quad := \quad POP(m);{}^{0,0,\mathtt{arg}}[\![r]\!]^{\emptyset}_{\emptyset,\emptyset}$$

- Compilation de deux patterns mis bout à bout.

$$\begin{array}{rcl} {}^{n,m,\mathtt{arg}}[\![P1P2]\!]^{\rho}_{\mathtt{Lbl},\mathtt{Cont}} & := & PUSHACCUFIELD\ 1; \uparrow \rho; \\ & & {}^{n,m+1,\mathtt{arg}}[\![P1]\!]^{\rho}_{\mathtt{Lbl},(P2,m+1,\mathtt{Cont})}; \end{array}$$

  Remarquer qu'on vérifie bien le point invariant suivant: au sommet (courant) de la pile, on met la valeur a matcher pour $P2$. On sauvegarde le "lieu" du sommet de la pile avec la variable $m+1$ que l'on met dans la nouvelle continuation. Ainsi, on pourra récurpérer la valeur à matcher plus tard (cf. infra).

  *Remarque.* Quelques questions sur le pushaccufield. Il est la pour mettre la dernière valeur de l'accumulateur sur la pile. Et les accumulateurs sont représentés dans le sens contraire [c : v3 v2 v1]

  Par exemple, si les accumulateurs étaient dans le bon sens, et si on voulait matcher $cP1P2P3$ avec $[c : v1v2v3]$ il faudrait faire un pushaccufield de 2, et non pas de 1 (comme c'est le cas ici). Le problème restant est qu'il faut quand-même retirer $v3$ de l'accumulateur, quoi qu'on fasse. Sinon ça va mal se passer (dans l'exemple précédent, il faudrait matcher ensuite c P1 P2 avec [c : v1 v2 v3], ce qui ne marche pas du tout. Étape à revoir, donc, une fois que j'en saurai plus sur les instructions PUSHACCUFIELD, et sur la structure des accumulateurs dans la ZAM.

Ici, j'ai fait une hypothese forte : on ne peut essayer de matcher qu'un accumulateur, et qui se trouve, qui plus est, dans le registre à ce moment là. Mais ca me semble raisonnable : que pourrait-on rencontrer d'autre ? La probleme vient peut-etre du fait que dans l'accu, il n'y a peut-etre pas assez de valeurs (y compris le nom de la constante de tete) — il faudrait faire un test de la taille de l'accu, avant d'essayer le PUSHACCUFIELD)

En effet, que peut-etre une valeur ? À part une abstraction ?

- Compilation des variables libres dans un pattern. Dans le cas où la continuation est un **TERME**

  On a terminé. On est arrivé au bout du pattern matching avec succès (une variable libre étant toujours intanciée correctement).

  Donc, il faut mettre les valeurs de $\rho$ sur le sommet de la pile, puis appeler le terme $T$, plus exactement son compilé. Mais auparavant (en fait, auparaprès), il faudra nettoyer la pile de tous les trucs crades qu'on a pu y mettre. Donc on rajoute un label de nettoyage, qui vire tout (y compris les arguments originels du pattern-matching). C'est là qu'on voit l'utilité de l'argument $m$ et `arg` ... passons aux détails.

  Étant donné que 0 est le dernier truc matché (dans $T$), c'est la valeur courante (celle qui est dans le registre ???), on la pousse .. .bref, a voir ! C'est pas clair de qui est ou dans ce bordel... en tous cas, la valeur d'indicde de DB 0 est dans le registre. Il faudrait la pousser, mais elle va devenir la valeur d'indice $n$, alors ! Donc, il faut la mettre dans $\rho$, et réempiler au dessus de la pile tout ce qu'il faut.

$$\begin{array}{rcl}
{}^{n,m,\texttt{arg}}[?]^{\rho}_{\texttt{Lbl},T} & := & PUSH; \uparrow \rho; \rho := 0.\rho; PUSHRA(\texttt{Lbl}(clean)); \uparrow^3 \rho; \\
& & \rho(n); PUSH; \uparrow (\rho); \rho(n-1); PUSH; \uparrow (\rho); \\
& & ...; \rho(0); PUSH; [T]; APPLY(n); \\
\texttt{Lbl}(clean) & : & POP(m+1+\texttt{arg}); return;
\end{array}$$

- Compilation des variables libres dans les patterns. Cas où on n'est pas arrivés au bout du pattern matching, c'est à dire si la continuation est de la forme $(P, m', \texttt{Cont})$

$$\begin{array}{rcl}
{}^{n,m,\texttt{arg}}[?]^{\rho}_{\texttt{Lbl},(P,m',\texttt{Cont})} & := & PUSH; \uparrow \rho; ACCES(m-m'+1); \\
& & {}^{n+1,m+1,\texttt{arg}}[P]^{\rho}_{\texttt{Lbl},\texttt{Cont}};
\end{array}$$

On récupère l'argument à matcher avec $P$, et il se trouve à la position $m - m' + 1$. Ceci est un invariant de la compilation (cf. le cas de la compilation de deux patterns $P1P2$). En fait, au sommet de la pile, il y aura toujours la valeur à matcher (est-ce vraiment une valeur ? je pense que oui, mais cela reste à voir). C'est à vérifier dans le calcul symbolique. Comme on ne peut pas metter de $\lambda$ dans le pattern, ça doit être bon.

D'où l'intérêt de sauvegarder la taille de la pile lorsqu'on créé la continuation.

- Compilation d'une variable liée. (Donc, qui a déjà été instanciée auparavant. Ici encore, deux cas possibles. Soit on a une continuation qui est un terme (fin du matching), soit une continuation qui est un pattern.

  Voyons d'abord le cas du terme.

$$\begin{array}{rcl}
{}^{n,m,\mathtt{arg}}[\![p]\!]^{\rho}_{\mathtt{Lbl},T} & := & PUSH; \uparrow \rho; \rho(p); ISEQUAL\ \mathtt{Lbl}(1); PUSHRA(\mathtt{Lbl}(clean)); \\
 & & \uparrow^3 \rho; \rho(n-1); PUSH; \uparrow \rho; \rho(n-2); PUSH; \uparrow \rho; \\
 & & ...; \rho(0); PUSH; [\![T]\!]; APPLY(n); \\
\mathtt{Lbl}(clean) & : & POP(m + arg); return; \\
\mathtt{Lbl}(1) & : & POP(m+1); JUMP\ \mathtt{Lbl};
\end{array}$$

On teste l'égalité entre les deux valeurs. La valeur courante, et l'instance de la variable (qui doit être une valeur aussi, je crois). On recherche pour l'instant seulement l'égalité "physique", car nous faisons du matching du premier ordre.

Pour cela, on pousse la valeur sur la pile, puis on met dans le registre la valeur a matcher, qui se trouve dans la pile à l'indice $\rho(p)$. Ensuite, on compare les deux valeurs :
la macro **ISEQUAL**, qui reste à définir, et qui devrait être un label spécialisé. Si ça matche, on continue. Si ca foire, on saute au label `Lbl`(1). On pourrait très bien faire l'inverse, c'est d'ailleurs peut-être mieux, car ça a plus de chances de foirer que de réussir ? Ça aussi, ça reste à voir.

Enfin, dans le cas où le matching réussit, il faut mettre les valeurs correspondantes au sommet de la pile. Et ne pas oublier de faire du nettoyage après. Peut-être serait-il mieux de faire le nettoyage avant, pour ne pas faire déborder la pile ? Mais dans ce cas, il faudrait créer une fermeture, virer tout ce qui dépasse de la pile, et puis remettre les valeurs à leur place, docn ça prendrait du temps supplémentaire.

D'ailleurs, question plus générale, où doit-on mettre les valeurs matchées ?

Commencer par nettoyer la pile nous permettrait de mieux gérer des fonctions sur-appliqués du style, avec la RR $f0 \to g$ et le code $f03$. Si on nettoie la pile après (comme on le fait maintenant, en fait), alors il faudrait relancer lévaluation du terme, en raison de la sur-application ?

Si on laisse le code comme il est, on aurait la chose suivante sur la pile: "$0''; ....; 0; 3; ...$ et on aurait peut-être des problèmes en ne nettoyant qu'après (à moins de mettre un RESTART dans le Lblclean ???).

Dans le cas où le matching foire, il faut sauter aux instruction en cas d'échec, qui se trouvent à l'endroit marqué par `Lbl`. Qui plus est, il faut nettoyer al pile de toutes les valeurs intermédiaires qu'on y a stocké. (mais PAS des arguments initiaux !)

- Compilation d'ne variable liée. Cas de la continuation-pattern.

$$\begin{array}{rcl} {}^{n,m,\texttt{arg}}[\![p]\!]^{\rho}_{\texttt{Lbl},(P,m',\texttt{Cont})} & := & PUSH; \uparrow \rho; \rho(p); ISEQUAL\texttt{Lbl}(1); ACCES(m-m'+1); \\ & & {}^{n,m+1,\texttt{arg}}[\![P]\!]^{\rho}_{\texttt{Lbl},\texttt{Cont}}; \\ \texttt{Lbl}(1) & : & POP(m+1); JUMP\texttt{Lbl}; \end{array}$$

Dans le cas où ça ne marche pas, c'est comme tout à l'heure. PAr contre, là où ça change, c'est dans le cas où ça marche.

- Compilation d'une constante "à matcher":

à peu près la même chose que dans le cas précédent, sauf qu'on n'a pas à aller chercher dans $\rho$. Voyons donc les deux cas possibles avec un peu plus de rapidité.

$$\begin{array}{rcl} {}^{n,m,\texttt{arg}}[\![c]\!]^{\rho}_{\texttt{Lbl},T} & := & PUSH(\text{"}c''); \uparrow \rho; IS-EQUAL\texttt{Lbl1}; PUSHRA(\texttt{Lbl}(clean)); \\ & & \uparrow^3 \rho; \rho(n-1); PUSH; \uparrow \rho; \rho(n-2); PUSH; \uparrow \rho; \\ & & ...; \rho(0); PUSH; [\![T]\!]; APPLY(n); \\ \texttt{Lbl}(clean) & : & POP(m+arg); return; \\ \texttt{Lbl}(1) & : & POP(m+1); JUMP\ \texttt{Lbl}; \end{array}$$

$$\begin{array}{rcl} {}^{n,m,\texttt{arg}}[\![c]\!]^{\rho}_{\texttt{Lbl},(P,m',\texttt{Cont})} & := & PUSH(<<c>>); \uparrow \rho; IS-EQUAL\ \texttt{Lbl}(1); ACCES(m-m'+1); \\ & & {}^{n,m+1,\texttt{arg}}[\![P]\!]^{\rho}_{\texttt{Lbl},\texttt{Cont}}; \\ \texttt{Lbl}(1) & : & POP(m+1); JUMP\texttt{Lbl}; \end{array}$$

Du moins, je crois que c'est comme ça que ça devrait marcher.

# References

[1] Grégoire, Benjamin, Thèse de Doctorat, 2003