Preuve d'un micro-compilateur pour le lambda-calcul non typé

16 avril 2021

L'objectif de ce projet consiste à formaliser et à prouver en Coq un micro-compilateur pour le lambda-calcul vers une machine à pile. L'ensemble du code et des preuves seront rédigées en Coq. Au cours de ce projet, nous aborderons tout d'abord la formalisation du langage source à l'aide d'indices de De Bruijn ainsi que sa sémantique, puis nous formaliserons le compilateur et enfin nous en prouverons la correction.
Note : les notions nécessaires qui ne sont pas présentées sur cette page le seront au tableau, au cours du projet.

1 Indices de De Bruijn et manipulation de λ-termes

Le λ-calcul est un modèle minimal des langages fonctionnels qui se prête très bien à l'étude de leur compilation. Il est défini par la grammaire ci-dessous :

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & | & \lambda x \cdot t & \text{fonction associant } t \text{ à } x \\ & | & t\ u & \text{application de la fonction } t \text{ à } u \end{array}$$

Ce langage permet donc de définir des fonctions et de les appliquer à des arguments. La sémantique de ce langage est définie par la notion de β-réduction qui consiste à évaluer l'application d'une fonction à son argument en effectuant une substitution de terme :

$$(\lambda x \cdot t)u \longrightarrow t[x \leftarrow u]$$

Lorsque l'on veut formaliser un langage réel comme Caml, on ajoute de nouvelles constructions (références, conditions, modules...), ainsi que des types, mais le principe reste le même.
La présentation classique du λ-calcul fait appel à des variables dont les noms sont des chaînes de caractères. On notera ainsi $\lambda x \cot x$ la fonction identité. L'inconvénient de cette notation vient d'une part du fait que l'on peut substituer tout autre nom à une variable liée sans changer le sens du terme (α-équivalence) et d'autre part du fait qu'il faille faire attention à renommer les variables de mêmes noms mais correspondant à des lieurs différents lors d'une substitution de terme.
Une solution alternative, plus proche d'une représentation en machine, fait appel à des entiers positifs, qui représentent le nombre de lieurs λ séparant l'occurence d'une variable de l'abstraction à laquelle elle est liée. Les noms de variables sous les λ deviennent alors inutiles (il s'agit toujours de la variable 0 en ce point). Ainsi, $\lambda x \cdot \lambda y \cdot yx$ sera représentée par le terme $\lambda \cdot \lambda \cdot (01)$. La notation d'un terme clos (c'est à dire dont toutes les variables sont liées à un λ;) avec de tels indices, appelés indices de De Bruijn est unique.

Travail demandé :

1. Définir les termes avec indices de De Bruijn en Coq.
2. Définir la notion de terme clos sous la forme d'un prédicat (astuce : il faut définir la notion plus générale de terme dont toutes les variables libres sont d'indice inférieur à n puis démontrer que si ce prédicat est vrai pour n il l'est aussi pour $n+1$; on notera $C[n](t)$ si t satisfait ce prédicat)

3. Définir une fonction de substitution sur les indices de De Bruijn (bien réflechir sur papier avant, car cette définition est difficile) et prouver que la substitution d'un terme dans un terme clos retourne ce dernier.
4. Définir une fonction de substitution plus générale prenant une liste de termes et effectuant plusieurs substitutions en parallèle ; pour cela définir

$$t[i \leftarrow u_0, i+1 \leftarrow u_1, \ldots, i+n-1 \leftarrow u_n]$$

(que l'on abrégera par $t[i \leftarrow u_0 \ldots u_n]$) Prouver, pour tout λ-termes $t, u_0, \ldots, u_n$ et tout entier i :
— $t[] = t$
— si $C[i](t)$, alors $t[i \leftarrow u] = t$
— si $\forall k \geq 1,\ C[i](u_k)$, alors $t[i \leftarrow u_0 \ldots u_n] = t[1+i \leftarrow u_1 \ldots u_n][i \leftarrow u_0]$

2 Sémantique des λ-termes

On peut à présent définir une sémantique à petit pas pour notre langage source. Cette sémantique décrit tout simplement la β-réduction, où l'argument d'une fonction est substitué au paramètre de celle-ci. Nous considérons ici cette relation comme étant non déterministe, c'est-à-dire que nous ne fixons pas d'ordre particulier à l'évaluation des β-réductions possibles, lorsqu'il y en a plusieurs (dans un langage réel, une stratégie précise est généralement choisie, comme par exemple l'appel par valeur, où l'on évalue toujours l'argument avant d'évaluer un appel de fonction l'une des extensions aborde ce problème).
La relation "t se réduit en u en une étape de β-réduction" notée $t \longrightarrow u$ peut être définie par les règles suivantes :
— **étape de calcul** : $(\lambda \cdot t)u \longrightarrow t[0 \leftarrow u]$
— **clôture par contexte** (i.e., réduction d'un sous terme) :
  — si $t \longrightarrow u$, alors $tv \longrightarrow uv$
  — si $t \longrightarrow u$, alors $vt \longrightarrow vu$
  — si $t \longrightarrow u$, alors $\lambda \cdot t \longrightarrow \lambda \cdot u$

La relation $\longrightarrow^*$ est alors définie par :
— $t \longrightarrow^* t$
— si $t \longrightarrow u$ et $u \longrightarrow^* v$, alors $t \longrightarrow^* v$

Travail demandé :

1. Formaliser la relation "t se réduit en u en une étape de β-réduction" ($\longrightarrow$) (on utilisera un prédicat inductif).
2. Formaliser la relation "t se réduit en u en un nombre quelconque d'étapes de réduction" ($\longrightarrow^*$).
3. Prouver que $\longrightarrow^*$ satisfait les mêmes propriétés de clôture par contexte que $\longrightarrow$.

3 La machine abstraite de Krivine (Krivine Abstract Machine)

Nous allons à présent nous intéresser au langage cible. Celui-ci consiste en trois instructions, et peut être décrit par la grammaire ci-dessous :

i	::=		instruction
	|	**Access** n	$onestunentier$
	|	**Grab**	
	|	**Push** c	
c	::=		bloc de code
	|	$i_0; \ldots; i_n$	

La machine de Krivine utilise deux piles :

— l'environnement qui conserve les termes correspondants aux variables libres du terme en cours d'évaluation ;

— la pile (stack) qui conserve les termes correspondants aux arguments des appels de fonctions en cours d'évaluation.

La pile consiste en une liste de paires constituée d'un fragment de code à exécuter et d'un environnement (la structure est donc récursive). Un état de cette machine correspond alors à un triplet de la forme

$$(c|e|s)$$

où c est un bloc de code, e un environnement et s une pile.
La sémantique de la machine de Krivine est décrite par les règles ci-dessous :

$$\begin{array}{rcll}
(\textbf{Access } 0; \ldots | (c_0, e_0).e | s) & \longrightarrow & (c_0|e_0|s) & \\
(\textbf{Access } n; c | (c_0, e_0).e | s) & \longrightarrow & (\textbf{Access } (n-1); c|e|s) & \text{si } n > 0 \\
(\textbf{Push } c'; c|e|s) & \longrightarrow & (c|e|(c', e).s) & \\
(\textbf{Grab}; c|e|(c_0, e_0).s) & \longrightarrow & (c|(c_0, e_0).e|s) &
\end{array}$$

Travail demandé :

1. Observer le fonctionnement de la machine abstraite "sur papier" ou à l'aide d'une mini-implémentation en Caml (utile pour se fixer les idées sur son comportement).
2. Formaliser les états de la machine abstraite de Krivine.
3. Formaliser sa sémantique (on pourra écrire une fonction prenant un état et renvoyant un type option, le terme "None" permettant de traiter les états pour lesquels il n'existe pas de transition.

4 Compilation

La fonction de compilation Comp est très simple :

$$\begin{array}{rcl}
Comp(\lambda \cdot t) & = & \textbf{Grab}; Comp(t) \\
Comp(tu) & = & \textbf{Push } (Comp(u)); Comp(t) \\
n & = & \textbf{Access } n
\end{array}$$

Travail demandé :

1. Définir la fonction de compilation en Coq.

5 Correction de la compilation

Pour établir la correction du compilateur, on va définir une traduction "à l'envers" τ qui permet d'associer à chaque état de la machine un programme "équivalent" :

$$\begin{array}{rcl}
\tau(\textbf{Access } n; c) & = & n \\
\tau(\textbf{Push } c'; c) & = & \tau(c)\tau(c') \\
\tau(\textbf{Grab}; c) & = & \lambda \cdot \tau(c) \\
\tau() & = & [] \\
\tau((c_0, e_0).e) & = & [0 \leftarrow \tau(c_0)[0 \leftarrow \tau(e_0)], u_1 \ldots u_n] \\
 & & \text{où } \tau(e) = [0 \leftarrow u_1 \ldots u_n] \\
\tau((c|e|(c_0, e_0).(c_1, e_1). \ldots .(c_n, s_n))) & = & ((\tau(c)[\tau(e)])(\tau(c_0)[\tau(e_0)]) \ldots (\tau(c_n)[\tau(e_n)]))
\end{array}$$

Une propriété importante de la machine de Krivine est qu'elle préserve au cours des calculs la clôture des termes, modulo cette traduction (autrement dit, elle ne perd pas le lien vers un sous-terme lorsque celui ci est placé sur la pile ou dans l'environnement). Pour cela, il faut définir ce qu'est une configuration "correcte". Une pile $(c_0, e_0).e$ est dite correcte si et seulement si e_0 et e sont correctes et $\tau(c_0)$ vérifie le prédicat de clôture $C[n]$ où n est la longueur de e_0. Un état $(c|e|s)$ est dit correct si et seulement si e est correct, $\tau(c)$ vérifie le prédicat de clôture $C[n]$ où n est la longueur de e et pour toute paire (c_k, e_k) dans la pile s, e_k est correct et c_k vérifie le prédicat de clôture $C[n_k]$ où n_k est la longueur de e_k. Cette propriété est fondamentale pour établir la correction de la compilation.

Travail demandé :

1. Définir τ.
2. Démontrer que lorsque l'on compile un terme puis qu'on applique la fonction τ, on retrouve le terme initial.
3. Démontrer que toute transition à partir d'un état correct conduit à un autre état correct (du point de vue de la clôture des termes dans l'environnement).
4. Démontrer que toute transition à partir d'un état correct correspond à $\longrightarrow$ modulo τ.
5. En déduire un théorême de correction de la compilation.