λ-calcul et compilation

Étienne Miquey
etienne.miquey@ens-lyon.fr

On va chercher dans ce TP à approcher (de loin) les idées de la compilation d'un language. On n'abordera ni l'analyse syntaxique ni l'analyse lexicale, et travaillerons pour cela avec un outil un peu particulier, le λ-calcul. Ce modèle de calcul, introduit dans les années 30 par Alonzo Church, a (entre autres) pour douces propriétés d'être Turing-complet (i.e. aussi puissant que n'importe quel autre language de programmation ou modèle d'ordinateur) et purement fonctionnel. Et en fait, on s'en sert même pour compiler certains languages fonctionnels.

1 Syntaxe

La syntaxe du λ-calcul est donnée par induction (ce qui ne devrait pas vous géner) :

$$\underset{(termes)}{T, U} \quad ::= \quad \underset{(variables)}{x} \quad | \quad \underset{(abstraction)}{\lambda x.(T)} \quad | \quad \underset{(application)}{TU}$$

et la seule règle de calcul (que l'on nomme la β-réduction) est donnée par

$$(\lambda x.T)u \longrightarrow_\beta T\{x := U\}$$

où $T\{x := U\}$ est le terme T dans lequel U remplace x.

Question 1. Définir formellement la substitution $T\{x := U\}$.

Question 2. Pour vérifier que vous avez tout compris, comment écririez-vous en CamL les λ-termes suivants : $\lambda x.x$, $\lambda x.(\lambda y.(yx))$, $\lambda b.(\lambda f_1.(\lambda f_2.((bf_1)xf_2)))$, $\lambda f.(\lambda g.(\lambda x.(f(gx))))$, $\lambda x.xx$

On définit donc le type suivant :

```
type lambda = Var of string | Abs of string * lambda | App of lambda * lambda;;
```

Question 3. Pour ne pas perdre les bonnes traditions, écrire une jolie fonction d'impression de λ-termes.

```
print_term : lambda -> unit
```

Nous allons chercher ici à calculer naïvement la réduction d'un λ-terme, avant de voir plus loin comment compiler un peu plus proprement la chose.

Question 4. En vous appuyant sur la définition formelle de la question 1, écrire une fonction `substitution` qui effectue la substitution d'une variable par un terme.

```
substitution : string -> lambda -> lambda -> lambda
```

Question 5. Écrire une fonction qui essaye de réduire un λ-terme.

```
reduce : lambda -> lambda
```

D'aucuns auront remarqué que programmer de la sorte est quelque peu hasardeux (aucune gestion possible des problèmes de terminaison ou d'explosion de la récursion). Et puis en plus, on triche, en se servant de la puissance de CamL.

2 Machine de Krivine

La machine de Krivine est une sorte de compilateur ad-hoc spécialement conçu pour le λ-calcul. Elle est constituée d'un terme de tête, d'une pile d'arguments et d'un environnement (qui fait le lien entre certaines variables et ce par quoi il faut les substituer), et fonctionne de façon impérative. Les règles sont les suivantes :

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

On définit donc les types suivants :

```
type closure = C of lambda * environment
and stack == closure list
and environment == (string * closure) list;;
```

```
type state == lambda * stack * environment;;
```

Vous avez bien entendu le droit, même si cela ne vous est pas explicitement demandé, d'écrire une fonction d'impression. De même, vous avez rudement intérêt à essayer de faire tourner la machine de Krivine à la main sur un petit exemple avant de vous lancer dans sa programmation.

Question 6. Ceci étant dit, écrire une fonction `access` qui permet d'accéder au contenu d'une variable dans un environnement.

```
access : string -> environment -> lambda * environment
```

Question 7. Écrire une fonction qui effectue un pas de calcul de la machine de Krivine, et lève une exception lorsque ceci n'est pas possible.

```
one_step : state -> state
```

Question 8. Enfin, écrire une fonction qui prendra en argument un terme et lancera le calcul dans une machine de Krivine initialement vide.

```
kam : lambda -> state
```

Essayez votre machine sur divers exemples (demandez m'en au besoin), et vérifiez que cela donne bien le résultat attendu ou cherchez pourquoi cela peut différer.

Lorsque l'on cherche à vérifier la correction de la machine, la méthode la plus courante est de disposer d'une sorte de fonction de décompilation, qui à chaque état machine associe le λ-terme correspondant. On vérifie alors qu'il y a correspondance entre les pas de calculs dans la machine et ceux par β-réduction.

Question 9. Écrire une première fonction qui, à un terme pris dans son environnement, associe le terme décompilé correspondant. On prendra bien garde aux variables d'abstraction non-encore liées dans l'environnement.

```
decomp_term : lambda -> environment -> lambda
```

Question 10. En réfléchissant à l'origine des variables sur la pile, écrire une fonction `decompilation` qui pour un état machine donné renvoie le λ-terme décompilé correspondant.

```
decompilation : state -> lambda
```

Question 11 (Bonus)**.** Tout un tas d'optimisations sont faisables plus ou moins facilement, pour gagner du temps et de l'espace. Des idées [1] ?

3 Assembleur

Nous allons aller encore plus loin dans la démarche de compilation, et désormais, nous n'exécuterons plus rien, mais traduirons dans un langage assembleur. Celui-ci sera par la suite traduit en langage machine puis pourra être interprété. Ainsi, nous ne pourrons plus utiliser quoi que ce soit de récursif, et devrons tout "impérativiser". Nous allons aussi nous abstraire des noms de variables, en se servant de label dans le code.

```
type instr = Grab | Push of int | Access of int | Label of int;;
```

Le schéma de compilation est le suivant (on note $[M]_\rho$ la compilation du terme M dans l'environnement ρ) :

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

Question 12. Écrire des fonctions `new_label` et `reset_label` qui maintiennent un compteur de label et permettront d'obtenir un nouveau label à tout instant.

```
new_label : unit -> int        reset_label : unit -> int
```

Désormais, nous ne pourrons plus nous servir de l'environnement (ce sera le luxe de l'interpréteur plus tard) pour connaître le contenu de nos variables. Nous allons donc juste rajouter les noms des variables dans l'environnement, ce qui suffit.

```
type env == string list;;
```

1. *Pensez notamment aux copies d'environnement et pointeurs à rallonge...*

Question 13. Écrire une fonction `make_label` qui créera et placera les nouveaux labels, et une fonction `compile` qui traduira un terme en une liste d'instruction. Si cela vous aide, vous pouvez utiliser une référence globale qui contiendra la liste des instructions

```
make_label : env -> lambda -> int
compile : env -> lambda -> instr list
```

Question 14. Écrire une fonction `cc` qui prend un λ-terme en entrée et renvoie la liste d'instructions correspondantes.

```
cc : lambda -> instr list
```

Pour finir, on peut encore peaufiner, pour traduire le langage assembleur en langage machine, en supprimant les labels (et en pointant explicitement là où il faut dans les `push`), puis écrire un interpréteur de cette séquence d'instructions

```
type instrM = GrabM | PushM of int | AccessM of int;;
```

Question 15. Écrire une fonction `assemble` qui prendra en entrée du code dans sa version avec label et le rendra en version machine.

```
assemble : instr list -> instrI list
```

Question 16 (Bonus)**.** Écrire une fonction qui effectuera l'interprétation d'une liste d'instructions en langage machine.