Fiche de TD no. 5

Le λ-calcul non typé

Exercice 1 : *Arbre abstrait, redex.* Dessinez l'arbre abstrait de chacun des λ-termes suivants et marquez ensuite tous les redexes dans un tel arbre :

1. $(\lambda x.\lambda y.((\lambda z.(z\ z))\ x))\ \hat{1}\ \texttt{SUCC}$;
2. $\texttt{SUCC}\ (\texttt{SUCC}\ \hat{1})$;
3. $\texttt{FIX}\ \texttt{SUCC}$;

où :

$$\hat{1} := \lambda f.\lambda x.(f\ x)\,, \qquad \texttt{SUCC} := \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)\,, \qquad \texttt{FIX} := \lambda g.(\lambda x.g\,(x\,x))\,(\lambda x.g\,(x\,x))\,.$$

Exercice 2. Voici un type de données `Haskell` adapté à coder les λ-termes du λ-calcul non typé :

```
type Variable = Int
data LambdaTerm = Var Variable
    | Appl LambdaTerm LambdaTerm
    | Abstr Variable LambdaTerm
```

1. Représentez le λ-terme $\lambda x_0.x_0x_1$ comme un valeur de type `LambdaTerm`.
2. Définissez une fonction

   ```
   redexes :: LambdaTerm -> [ LambdaTerm ]
   ```

   qui calcule tous les redexes dans un λ-terme passé en paramètre.
3. Modifiez cette définition de façon que le premier élément de la liste retournée soit le redex choisi par la stratégie d'évaluation par nom (resp. par valeur).

Exercice 3. Considérez ces trois λ-termes :

$$\texttt{TRUE} := \lambda x.\lambda y.x\,, \qquad \texttt{FALSE} := \lambda x.\lambda y.y\,, \qquad \texttt{IFTHENELSE} := \lambda p.\lambda a.\lambda b.p\,a\,b\,.$$

Soient t_1, t_2 deux λ-termes arbitraires ; montrez que

$$\texttt{IFTHENELSE}\ \texttt{TRUE}\ t_1\ t_2\ \xrightarrow{*}_\beta\ t_1 \quad \text{et} \quad \texttt{IFTHENELSE}\ \texttt{FALSE}\ t_1\ t_2\ \xrightarrow{*}_\beta\ t_2\,.$$

Afin d'éviter le renommage des variables lors des substitutions, on pourra supposer que t_1 et t_2 ne contiennent pas les variables x, y, a, b, p.

Exercice 4. Rappel : on déclare deux λ-termes t_1, t_2 équivalents (et on écrit $t_1 \equiv t_2$) si $t_i \xrightarrow{*}_\beta t'$, pour $i = 1, 2$ et un quelque λ-terme t'.
Considérez le λ-terme suivant :

$$\texttt{FIX} := \lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$$

1. Soit t un terme arbitraire (on pourra supposer que t ne contient pas les variables x et g) ; montrez qu'il existe un terme t' tel que

   $$\texttt{FIX}\ t\ \xrightarrow{*}_\beta t' \quad \text{et} \quad \texttt{FIX}\ t\ \xrightarrow{*}_\beta\ t\ t'\,.$$

2. Argumentez ainsi que

   $$\texttt{FIX}\ t \equiv t\ (\texttt{FIX}\ t)$$

   de facon qu'on peut considérer $\texttt{FIX}\ t$ comme un point fixe de la "fonction" t.

Exercice 5 : *Stratégies d'évaluation.* Utilisez les différentes stratégies—stratégie par l'intérieur (appelée aussi *ordre applicatif*), stratégie par l'extérieur (appelée aussi *ordre normal*), stratégie « *call by value* » et stratégie « *call by name* »— pour évaluer le premier termes proposé à l'exercice 1. S'il vous reste du temps, évaluez (avec les quatre stratégies) aussi le deuxième et troisième terme de cet exercice.

Évaluation, en Haskell

Exercice 6. Utilisez d'abord la stratégie *call-by-value* et ensuite la stratégie *call-by-name* pour évaluer les expressions suivantes :

```
(\b -> \x -> \y -> if b then x else y) (True && True) [] (replicate 2 '*')
take 2 (repeat '*')
add (Succ Zero) Zero
(\x -> \y -> x) 5 omega where omega = omega + 1
```

(Rappelez éventuellement les définitions de `replicate`, `repeat`, `add`, `omega`.)

Exercice 7. Considérez les définitions suivantes de la fonction `fib :: Int -> Int` qui associe à un entier n le n-ème nombre de Fibonacci :

```
fib1,fib2,fib3 :: Int -> Int

fib1 0 = 1
fib1 1 = 1
fib1 n = fib1 (n-1) + fib1 (n-2)

fib2 0 = 1
fib2 1 = 1
fib2 n = fibliste !! (n -1) + fibliste !! (n-2)
fibliste = map fib2 [0..]

fib3 n = fib3acc n 1 1
  where
    fib3acc 0 n m = n
    fib3acc k n m =  fib3acc (k -1) m (m + n)
```

Quelle est, à votre avis, la plus (resp. la moins) performante ? Justifiez votre réponse.

Exercice 8. Démontrez que les fonctions `fib1` et `fib3` depuis l'exercice 7 calculent la même suite infinie de nombres.

Exercice 9. Considérez la script suivant :

```
import Prelude hiding (seq)

seq :: Int -> a -> a
seq n x = if n == 0 then x else x

sum1 n []  = 0
sum1 n (x:xs) = sum1 (n+x) xs

sum2 n [] = 0
sum2 n (x:xs) = let n' = n+x in
  n' `seq` sum2 n' xs

bignumber = 1000000
test1 = sum1 0 [1..bignumber]
test2 = sum2 0 [1..bignumber]
```

Quel est, selon vous, le test (parmi `test1` et `test2`) qui utlise moins de mémoire lors de son évaluation ?