UFR 919 Sciences de l'ingénieur
Master Informatique – Spécialité STL
Analyse de Programmes et Sémantique (APS)

TD5 — λ-calcul et Scheme

Jacques Malenfant, Olena Rogovchenko

1 Le λ-calcul

1.1 Syntaxe des termes

Les termes suivants sont-ils syntaxiquement corrects ? Si oui, décomposez-les en sous-termes et en délimitant toutes les abstractions et applications.

1. $\lambda z.z\ y\ y$
2. $(x\ y).\lambda xy$
3. $(\lambda x.y)\ x\ x$
4. $\lambda x.x\ y\ \lambda t.t\ x$

1.2 Variables libres et substitution

On a vu que l'ensemble $VL(t)$, les variables libres d'un λ-terme t est défini par récurrence de la manière suivante :

— si x est une variable alors $VL(x) = \{x\}$
— si u et v sont des λ-termes alors $VL(u\ v) = VL(u) \cup VL(v)$
— si x est une variable et u un λ-terme alors $VL(\lambda x.u) = VL(u) \backslash \{x\}$

Un λ-terme est dit clos s'il ne contient aucune variable libre.

Peut-on donner une définition similaire pour les variables liées d'un λ-terme ? Pourquoi ?

Calcul des variables libres

En détaillant l'application de la définition, donner l'ensemble de variables libres pour les expressions suivantes :

1. $\lambda y.y\ y$
2. $\lambda x.x\ y$
3. $\lambda c.c\ (\lambda ab.a)$
4. $\lambda c.c\ (\lambda z.a\ z\ c)z$

Est-ce que les termes 1 et 2 sont α-congruents ?

Calcul des substitutions

En suivant la définition du cours, appliquer les substitutions indiquées, en renommant les variables lorsque c'est nécessaire :

1. $(\lambda y.y\ y)[y := x]$
2. $(\lambda x.x\ y)[y := x]$
3. $(\lambda c.c\ (\lambda z.a\ z\ c)\ z)[z := c]$

1.3 Réduction

Réduire ces termes en appliquant d'abord la stratégie de réduction en ordre normal (par l'extérieur) et ensuite la stratégie de réduction en ordre applicatif (par l'intérieur) :

1. $(\lambda x.x)\ (\lambda yzu.y\ (z\ u))\ a\ b\ c$
2. $(\lambda fx.(f\ (f\ x)))\ x$
3. $(\lambda x.x\ x\ x)\ ((\lambda x.x)\ \lambda x.y)$
4. $(\lambda xy.y)\ ((\lambda y.y\ y)\ (\lambda x.x\ x))\ a$

Est-ce que l'on obtient le même résultat avec les deux stratégies de réduction ?

1.4 Encodage de fonctions en λ-calcul

Logique et prédicats

On définit les fonctions :

— $\mathbf{A} \equiv \lambda cab.c\ a\ b$
— $\mathbf{B} \equiv \lambda xy.x$
— $\mathbf{C} \equiv \lambda yx.x$

Réduire les termes suivants :

1. **A B**
2. **A C**
3. **B** M où M désigne un terme quelconque
4. **C** M où M désigne un terme quelconque

Qu'en concluez-vous sur la nature des termes **A**, **B** et **C** par rapport à la logique ?

Donner des définitions pour les opérateurs logiques **and**, **or** et **not**. (On pourra se servir des fonctions définies précédemment).

Les entiers naturels

Church a proposé un encodage des entiers naturels par la composition de fonction. Ainsi, l'entier n est représenté par la composition d'une fonction f n fois avec elle-même. On a donc :

$$
\begin{array}{ccl}
0 & \equiv & \lambda f x . x \\
1 & \equiv & \lambda f x . f\ x \\
2 & \equiv & \lambda f x . f\ (f\ x) \\
3 & \equiv & \lambda f x . f\ (f\ (f\ x)) \\
\ldots & \ldots & \ldots
\end{array}
$$

Étant donné cet encodage, donner une définition à la fonction successeur **SUCC**.

Donner une définition pour la fonction d'addition **PLUS**.

Enfin donner une définition à la fonction de multiplication **MULT**.

Les paires

On encode un couple par le terme

$$\textbf{PAIR} = \lambda x y f . f x y$$

Donner les définitions de **FIRST** et **SECOND** qui pour tous les termes M, N se réduisent de la manière suivante :

— **FIRST** (**PAIR** $M\ N$) $\Rightarrow\ M$
— **SECOND** (**PAIR** $M\ N$) $\Rightarrow\ N$

2 Premiers pas en Scheme

En TP, on utilise **Gambit**, un compilateur Scheme-vers-C conforme à R5RS. Pour lancer Gambit :

— l'interprète : `/usr/local/gambc-v4_6_0-devel/bin/gsi`
— le compilateur : `/usr/local/gambc-v4_6_0-devel/bin/gsc`

En vous aidant de la documantation Scheme qui vous est fournie sur le site de l'UE, répondez aux questions suivantes.

Q1. Le terme `(let ((x 1) (y (* 2 x))) (+ x y))` est-il valide en Scheme ? Pourquoi ?

Q2. Écrire en Scheme la fonction `factorielle`.

Q3. Les prédicats en Scheme : un prédicat est une fonction ayant pour valeur un booléen. Par convention ils se terminent par un ?. Évaluer les prédicats suivants `(number? 5)` et `(procedure? +)` et définir le predicat `not-zero?`.

Q4. Écrire en Scheme le λ-terme $((\lambda a.\lambda b. +\ a\ b)\ 1\ 2)$.

Implantation du λ-calcul en Scheme

La grammaire des λ-termes purs est :

$$\mathit{term} \quad ::= \quad v \ \mid\ (\mathit{term}\ \mathit{term}) \ \mid\ \texttt{lambda}\ v\ \mathit{term}$$

Ce qu'on peut implanter en Scheme en prenant les symboles pour représenter les variables, les listes à deux membres pour représenter les applications et les listes à trois membres débutant par le symbole `lambda` pour représenter les abstractions. On peut alors se donner des fonctions de manipulation de cette représentation des arbres de syntaxe abstraite des λ-termes :

```
(define (make-var symb) symb)
(define (var? term)     (symbol? term))

(define (make-abstraction var body) (list 'lambda var body))
(define (get-formal abstraction)    (cadr abstraction))
(define (get-body abstraction)      (caddr abstraction))
(define (abstraction? term)
  (and (list? term) (equal? (car term) 'lambda)))

(define (make-application operator operand) (list operator operand))
(define (get-operator application)          (car application))
(define (get-operand application)           (cadr application))
(define (application? term)
  (and (list? term) (not (abstraction? term))))
```

Q1. Définissez-vous un type abstrait ensemble (sets), en vous appuyant sur les listes de Scheme avec des fonctions `make-empty-set`, `add2set`, `list2set`, `set?` et `empty-set?`, puis vous définirez les opérations `in?`, `union`, `intersection` et `difference`.

Q2. Implantez la fonction `free-variables` qui prend un *lambda*-terme et retourne l'ensmeble de ses variables libres.

Q3. Implantez la fonction `substitution` qui prend un λ-terme `t1`, une variable `v` et un second λ-terme `t2` et qui retourne le λ-terme `t` où la variable `v` a été substituée par `t2`.

Q4. Implantez les fonctions `reduce-normal` et `reduce-applicative` qui prennent un λ-terme et le réduise respectivement en ordre normal et en ordre applicatif.
Indication : il vous sera utile de définir des fonctions `redex?` et `has-redex?` qui détermine si un terme est un β-redex ou s'il contient un β-redex.