Examen du cours PROG1

12 décembre 2022

Durée : 2 heures.
À réaliser sans document ni machine.

1 Lambda-calcul

Question 1 Dessiner l'arbre de syntaxe abstraite de chacun des λ-termes suivants, et indiquant chaque β-redex (en surlignant, sous-lignant, ou colorant) :

— $(\lambda x.\ x\ x)\ x$
— $\lambda x.\lambda y.\ x\ (\lambda z.\ y\ z)\ y$

Question 2 Énoncer les théorèmes de préservation du typage [1] et de normalisation faible pour le λ-calcul simplement typé.

Question 3 On enrichit le λ-calcul simplement typé avec un point fixe, en ajoutant un terme spécial fix, une nouvelle règle de réduction

$$\mathsf{fix}\ (\lambda f.\lambda x.M)\ N \to M[x := N, f := (\mathsf{fix}\ (\lambda f.\lambda x.M))]$$

et une nouvelle règle de typage :

$$\frac{}{\Gamma \vdash \mathsf{fix} : ((T_1 \to T_2) \to T_1 \to T_2) \to T_1 \to T_2}$$

Parmi les deux théorèmes précédents, lesquels restent vrai dans cette extension ? une réponse négative devra être justifiée par un contre-exemple, mais une réponse positive n'aura pas besoin d'être justifiée.

2 Promenons-nous dans un arbre

On considère des arbres binaires définis comme suit :

```
type tree = Node of tree*tree | Leaf of int
```

On se donne ensuite une notion de chemin dans un arbre, définie par le type suivant :

1. Aussi appelé réduction du sujet.

```
type instruction = Left | Right | Up | Check of int
type path = instruction list
```

Un chemin est une liste d'instructions à suivre. L'instruction `Left` indique de descendre dans le sous-arbre gauche, et `Right` fait de même à droite. L'instruction `Up` indique de remonter au noeud père. Enfin, `Check i` indique qu'il faut vérifier qu'on est sur une feuille étiquetée `i`. Il n'est pas toujours possible de suivre un chemin : chaque instruction peut échouer sur certains noeuds.

Par exemple, si l'on part de la racine de l'arbre `Node (Node (Leaf 1, Leaf 2), Leaf 3)` on peut suivre les chemins `[Right; Check 3]` et `[Left; Left; Check 1; Up; Right]` mais pas `[Left; Left ; Check 1; Up; Right ; Check 1]` car le deuxième `Check 1` échoue sur `Leaf 2`.

Question 1 Pour tout arbre `t` il existe un chemin `p` qui, partant de la racine, parcourt intégralement l'arbre, vérifiant les étiquettes des feuilles, pour finalement retourner à la racine. Il doit être possible de suivre ce chemin à partir de la racine de `t`, mais pas à partir de la racine d'un arbre différent. Implémenter une fonction (a priori pas récursive terminale) calculant un tel chemin :

```
val path_of_tree : tree -> path
```

On souhaite maintenant implémenter une fonction qui permet de suivre un chemin à partir de la racine d'un arbre. On suppose déclarée une exception qu'on utilisera pour signaler les instructions invalides :

```
exception Error
```

Question 2 Comme un chemin peut descendre dans un sous-arbre puis en ressortir via des instructions `Up`, il nous faut généraliser la spécification afin de pouvoir obtenir une solution récursive. Implémenter une fonction `follow_path : tree -> path -> path` qui suit un chemin dans un arbre et renvoie le reste du chemin à exécuter après une éventuelle sortie de la racine par une instruction `Up`. Si le chemin se termine dans l'arbre, la liste vide est simplement retournée. Par exemple, on a :

```
let () =
  let p = [Check 1 ; Up ; Right ; Check 2] in
  assert (follow_path (Leaf 1) p = [Right ; Check 2]) ;
  assert (follow_path (Node (Leaf 1, Leaf 2)) (Left :: p) = [])
```

De plus, pour tout arbre `t` on aura `follow_path t (path_of_tree t)= []`.

Question 3 Transformer la fonction précédente pour procéder par passage de continuation (CPS, *continuation passing style*). La fonction résultante devrait avoir le type suivant, et être récursive terminale :

```
val follow_path_cps : tree -> path -> (path -> 'a) -> 'a
```

On devrait de plus avoir, pour tous `t` et `p`, `follow_path_cps t p (fun x -> x)= follow_path t p`.

Question 4 Donner une troisième version de cette fonction, récursive terminale et n'utilisant pas l'ordre supérieur [2]. Vous pouvez procéder par défonctionnalisation, mais une solution obtenue de façon moins systématique sera acceptée sans pénalité. Le type est libre, mais veillez à ce que l'utilisation d'éventuels arguments supplémentaires soit claire, en indiquant au moins avec quelles valeurs initiales la fonction devra être appelée pour retrouver le calcul de `follow_path t p`.

3 Gestion mémoire

Question 1 On considère la fonction suivante :

```
let f n =
  let l = List.init n (fun i -> i) in
  List.map (fun x -> i,i) l
```

Combien de blocs (en fonction de `n`) sont alloués lors d'un appel à `f n` ? Combien peuvent être recyclés par le garbage collector ? (On suppose que la valeur retournée par la fonction est vivante.) Proposez une amélioration de cette fonction.

Question 2 On considère le programme suivant :

```
let rec iter f l = match l with
  | [] -> ()
  | hd::tl -> f hd ; iter f tl

let () =
  iter (fun i -> Printf.printf "%d\n" i) (List.init 5 (fun i -> i))
```

À chaque appel de `printf`, combien de blocs sont vivants en mémoire, en fonction de `i` ?

Question 3 Pour chacune des deux modifications suivantes du programme, est-ce que la réponse à la question précédente change ? En cas de changement indésirable, proposez une amélioration de la fonction.

(a) On considère d'abord une variante par passage de continuations :

```
let rec iter_cps f l cont = match l with
  | [] -> cont ()
  | hd::tl -> f hd (fun () -> iter_cps f tl cont)
let () =
  iter_cps
    (fun x cont -> Printf.printf "%d\n" x ; cont ())
    (List.init 5 (fun i -> i))
    (fun () -> ())
```

2. Votre fonction ne peut donc pas prendre une fonction en argument, ni directement ni indirectement (pas de liste de fonctions, par exemple). Plus généralement, aucune clôture ne devra être allouée lors de l'exécution de votre fonction.

(b) On considère ensuite une variante où `iter` rattrape certaines erreurs levées par `f` pour continuer malgré cela l'itération :

```
let rec iter f l = match l with
  | [] -> ()
  | hd::tl ->
    try f hd ; iter f tl with
    | Failure _ -> iter f tl
let () =
  iter (fun i -> Printf.printf "%d\n" i) (List.init 5 (fun i -> i))
```

4 Modules

On souhaite construire, à partir d'une représentation quelconque des entiers relatifs, une représentation des rationnels. Pour cela, on va utiliser le système de modules d'OCaml.

On considèrera une implémentation des entiers satisfaisant la signature suivante :

```
module type INT = sig
  type int
  val zero : int                (* neutre de l'addition        *)
  val one : int                 (* neutre de la multiplication *)
  val lt : int -> int -> bool   (* strictly [l]ess [than]      *)
  val add : int -> int -> int   (* addition                    *)
  val opp : int -> int          (* inverse pour l'addition     *)
  val mult : int -> int -> int  (* multiplication              *)
  val div : int -> int -> int   (* division                    *)
  val gcd : int -> int -> int   (* greatest common divisor     *)
end
```

Dans tout cet exercice, on exige de façon inhabituelle que l'égalité structurelle (le `(=)` d'OCaml) sur les types abstraits corresponde à l'égalité naturelle sur ces objets. Ainsi, une implémentation de `INT` pourrait représenter des entiers par une suite de chiffres et un signe, mais il faudrait éviter que deux suites de chiffres corresponde au même nombre.

Voici la signature attendue pour les rationnels :

```
module type RAT = sig

  (** Type abstrait dont les valeurs représentent des nombres
      rationnels. *)
  type rat

  (** Type abstrait des entiers sur lesquels sont construits
      les rationnels. *)
  type int

  (** Conversion d'un entier en rationnel. *)
  val of_int : int -> rat
```

```
  (** Addition. *)
  val add : rat -> rat -> rat

  (** Multiplication. *)
  val mult : rat -> rat -> rat

  (** Inversion.
      Lève l'exception [Division_by_zero] si le rationnel
      à inverser à nul. *)
  val inv : rat -> rat

end
```

Question 1 Définir un module `MyInt` satisfaisant la signature `INT with type int = Int.t` (le type `Int.t` est un autre nom pour le type `int` habituel d'OCaml) en omettant l'implémentation de `gcd`, qu'on pourra néanmoins utiliser dans la suite.

Question 2 Remplir la définition suivante, où l'on implémente un foncteur construisant des rationnels à partir d'une représentation arbitraire des entiers :

```
module MakeRat (I:INT) : RAT = struct
  type rat = I.int*I.int
  type int = I.int
  (* ... *)
end
```

Veillez à bien expliciter quels invariants sont maintenus sur les objets de type `rat`. Ceux-ci doivent être des paires, mais l'égalité structurelle sur `rat` doit correspondre à l'égalité des rationnels représentés.

Question 3 On envisage maintenant d'instancier `MakeRat` sur `MyInt` et de l'utiliser :

```
module R = MakeRat(MyInt)
let () =
  let two = MyInt.add MyInt.one MyInt.one in
  let half = R.inv (R.of_int two) in
  assert (R.of_int MyInt.one = R.add half half)
```

Le compilateur signale cependant une erreur de type sur ce programme :

```
File "exam_dec_2022.ml", line 175, characters 29-32:
175 |   let half = R.inv (R.of_int two) in
                                 ^^^
Error: This expression has type MyInt.int = int
       but an expression was expected of type R.int = MakeRat(MyInt).int
```

Expliquez et corrigez cette erreur.