Examen du cours PROG1

16 décembre 2021

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

1 Modules

On considère le programme suivant, où l'on évite les notations `a.(i)` et `a.(i) <- v` en explicitant leur signification en termes d'appels de fonctions du module `Array`, i.e. respectivement `Array.get a i` et `Array.set a i v` :

```
let f a =
  let n = Array.length a in
  let a' = Array.init n (fun i -> i, Array.get a i) in
  Array.sort (fun x y -> Stdlib.compare (snd x) (snd y)) a' ;
  for i = 0 to n-1 do
    Array.set a i (snd (Array.get a' i))
  done ;
  Array.init n (fun i -> fst (Array.get a' i))
```

On rappelle que :

- `Array.init l elt` alloue un nouveau tableau de longueur `l` avec `elt i` dans la case d'index `i` ;
- `Stdlib.compare : 'a -> 'a -> int` compare deux valeurs [1], renvoyant `0` quand elles sont égales, `1` quand la première est supérieure à la seconde, et `-1` sinon ;
- `Array.sort cmp arr` trie en place le tableau `arr` pour le rendre croissant au sens de `cmp`.

Question 1 Quel est le type de `f` ? Que fait cette fonction ? Que se passerait-t-il si on utilisait `Stdlib.compare` au lieu de `fun x y -> Stdlib.compare (snd x)(snd y)` lors de l'appel à `Array.sort` ?

Question 2 Donner la signature `S` que le module `Array` doit réaliser pour que cette fonction soit typable. On visera une signature minimale en termes de fonctionalités et d'information sur les types.

Question 3 On souhaite compter le nombre de comparaisons effectuées lors du tri, sans changer le code de la fonction `f`, mais en re-définissant le module `Array`. Remplir le patron de programme suivant à cet effet. Le nombre de comparaisons effectuées devra être affiché à l'issue de chaque tri.

```
module Array : S = struct
  (* À vous de jouer. *)
end
let f () =
  (* Comme ci-dessus. *)
```

1. Cette fonction renvoie une exception sur certaines valeurs, e.g. les fonctions, mais cet aspect peut être ignoré.

Question 4 On souhaite pouvoir appliquer la transformation précédente à n'importe quel module de signature `S`, et pas seulement au module `Array` de la librairie standard. Proposer une solution.

Question 5 Comment rendre la fonction `f` indépendante de l'implémentation particulière de `Array` ? Proposer une solution permettant l'utilisation de cette fonction pour des implémentations arbitraires de `S`.

Question 6 On considère une mauvaise solution à la question précédente, d'abord sur un programme plus simple qui utilise le tri pour extraire le maximum d'un tableau d'entiers :

```
(* Version initiale *)
let g a =
  let a' = Array.init (Array.length a) (fun i -> -(Array.get a i)) in
  Array.sort Stdlib.compare a' ;
  -(Array.get a' 0)

(* Version abstraite *)
let g' length init get set sort a =
  let a' = init (length a) (fun i -> -(get a i)) in
  sort Stdlib.compare a' ;
  -(get a' 0)

let () =
  let a = [| 1 ; 3 ; 0 |] in
  assert (3 = g a) ;
  assert (3 = g' Array.length Array.init Array.get Array.set Array.sort a)
```

Expliquer pourquoi cette approche ne convient pas pour notre fonction `f`, i.e. expliquer ce qui ne va pas dans le programme suivant :

```
let f length init get set sort a =
  let n = length a in
  let a' = init n (fun i -> i, get a i) in
  sort (fun x y -> Stdlib.compare (snd x) (snd y)) a' ;
  for i = 0 to n-1 do
    set a i (snd (get a' i))
  done ;
  init n (fun i -> fst (get a' i))
```

2 Lecture de lignes

On considère le programme suivant :

```
let rec read () = try read_line () :: read () with End_of_file -> []
```

La fonction `read_line : unit -> string` de la librairie OCaml renvoie la prochaine ligne lue sur l'entrée standard (`stdin`) et lève l'exception `End_of_file` en cas de fin de fichier.

Expliquer ce que fait la fonction `read`. Expliquer pourquoi l'exécution de cette fonction provoque un déborde de pile (`Stack_overflow`) quand trop de lignes sont présentes sur l'entrée standard. Corriger ce problème en réécrivant le programme, mais en restant dans un style fonctionnel et récursif — références et boucles interdites.

3 Séparation

Si `a` et `b` sont deux valeurs de même type, on dit qu'une fonction `f` les sépare quand `f a` s'évalue en `true` et `f b` s'évalue en `false`. Par exemple, `fun v -> v=0` sépare `0` et `42`.

Pour les besoins de cet exercice on déclare deux exceptions :

```
exception E
exception F of int
```

Pour chaque paire `aN`, `bN` ci-dessous, donner le type des deux valeurs [2] puis donner une fonction `fN` qui les sépare. Attention : on veut avoir `f a = true && f b = false` et pas seulement `f a <> f b` !

On rappelle qu'en OCaml, le type des exceptions s'appelle `exn`.

1. ```
   let a1 = fun ()-> true
   let b1 = fun ()-> false
   ```
2. ```
   let a2 = fun f x y -> f x
   let b2 = fun f x y -> f y
   ```
3. ```
   let a3 = fun ()-> ()
   let b3 = fun ()-> raise E
   ```
4. ```
   let a4 = fun f e -> raise e
   let b4 = fun f e -> raise (f e)
   ```
5. ```
   let a5 = fun f -> try f (); raise E with F _ -> true
   let b5 = fun f -> try f (); raise E with F 0 -> true
   ```
6. ```
   let a6 = fun p -> let x,y = p ()in fun ()-> (x,y)
   let b6 = fun p -> p
   ```

4 Style par passage de continuations

On considère la fonction suivante :

```
type 'a t = Node of 'a t * 'a t | Leaf of 'a
let rec parity t = match t with
  | Leaf i -> Leaf (i mod 2 = 0)
  | Node (l,r) -> Node (parity l, parity r)
```

Question 1 Que fait cette fonction et quel est son type ?

Question 2 Passer la fonction en style par passage de continuation (CPS). Préciser le type de la fonction transformée.

Question 3 Défonctionnaliser la fonction précédente. On suivra une approche systématique en s'abstenant de simplifier le code résultant.

2. Si dans certains cas un des deux types est plus général que l'autre, la fonction de séparation fonctionnera bien sûr sur le type le plus restrictif.

5 Hash-consing

Sur le type des arbres binaires de l'exercice précédent, deux sous-arbres structurellement équivalents peuvent être physiquement différents : en effet, chaque sous-expression `Node (l,r)` donne lieu à l'allocation d'un nouveau bloc représentant ce noeud. Ce type de redondance cause une utilisation mémoire inutilement élevée, et le besoin d'utiliser l'égalité structurelle est aussi coûteux en temps puisqu'elle nécessite le parcours des arbres comparés.

Le *hash-consing* est une technique de programmation qui vise à assurer que l'égalité physique et l'égalité structurelle coïncident, ce qui permet d'éviter les problèmes mentionnés précédemment.

Nous étudions une modification du type des arbres précédent, sur lequel nous allons mettre en place un mécanisme de hash-consing. Afin de garder un exercice assez simple, nous n'utiliserons pas de tables de hash, mais de simples listes.

```
type 'a t = Node of 'a t * 'a t | Leaf of 'a

let cache = ref []

let unique t =
  try List.find (fun t' -> t = t') !cache with
    | Not_found -> cache := t :: !cache ; t

let leaf i = unique (Leaf i)

let node l r  = unique (Node (l,r))
```

Deux arbres créés via les fonctions `leaf` et `node` ci-dessus sont tels qu'on a `t = t'` si et seulement si `t == t'`.

Question 1 En supposant que tous les arbres sont créés via `leaf` et `node`, expliquer pourquoi le test d'égalité fait dans l'appel à `List.find` est inutilement coûteux, et proposer une version améliorée de ce test.

Question 2 Comment assurer que le programmeur ne crée pas d'arbres via l'utilisation directe des constructeurs `Leaf` et `Node` ?

Question 3 Expliquer pourquoi cette technique empêche le travail du ramasse-miettes (*garbage collector*) et proposer une amélioration du système qui remédie à ce problème.