

Examen du 17 juin 2008

Les notes de TD manuscrites de cette année sont les seuls documents autorisés.

Veillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse. Le barème est indicatif.

Important. Les types et l'utilisation des fonctions de la bibliothèque standard d'Ocaml mentionnées (ou pouvant être utiles) dans les questions suivantes sont donnés à la fin de ce document.

1 Typage (5 points)

Question 1.1 Donner le type de `r` après chaque déclaration.

```
let r = ref [([], [])]  
let _ = r := List.map (fun (x,y) -> (y,x)) !r
```

Question 1.2 Les fonctions `f1`, `f2` et `f3` suivantes sont-elles bien typées? Si oui, donner leur type, sinon préciser pourquoi

```
let f1 x y z = if x then y := x else z := !y
```

```
let rec f2 x = f2 (!x+1)
```

```
let rec f3 l =  
  match l with  
  | [] -> false  
  | [(x,_)] -> x  
  | (_,y)::s -> let z = f3 s in y:=z ; z
```

2 Évaluation (5 points)

Question 2.1 Étant donné le type `t`, l'exception `Goto` et la fonction `mystere` suivants :

```
type e = E1 | E2 | E3  
exception Goto of e
```

```
let rec mystere x = try  
  if x=0 then raise (Goto E1) ;  
  if x mod 2 = 0 then raise (Goto E2);  
  raise (Goto E3)  
with  
  Goto (E1|E3) -> mystere (x+1)  
  | Goto E2 -> x*10
```

Donner (en le justifiant) la valeur retournée par l'expression `mystere 0`

Question 2.2 Donner (en le justifiant) les valeurs de chaque case du tableau `m` après la dernière expression.

```
let m = Array.make 4 (ref 2)
let _ =
  for i = 0 to 3 do
    m.(i) := i
  done
```

Question 2.3 Étant donnée la fonction `mystere` suivante :

```
let mystere f x = let y = f x in y / !x;;
```

Donner la valeur retournée par l'expression `mystere (fun x -> x := !x+1; !x) (ref 0)`

3 Programmation : File à changement de priorité (10 points)

Le but de cette partie est de développer une structure de données pour représenter une file à priorités sur laquelle on peut effectuer les trois opérations suivantes :

- ajouter un élément dans la file avec une certaine priorité;
- extraire de la file l'élément le plus prioritaire;
- rendre plus prioritaire un élément déjà dans la file.

Le type abstrait de cette structure de données est défini par la signature `FILE` suivante :

```
module type FILE = sig
  type pointeur
  type 'a t
  val creer : int -> 'a t
  val ajouter : 'a -> int -> 'a t -> pointeur
  val rendre_plus_prioritaire : 'a -> pointeur -> int -> 'a t -> unit
  val extraire_le_plus_prioritaire : 'a t -> 'a
end
```

Cette signature définit deux types abstraits : `'a t` pour les files à priorités contenant des éléments de type `'a` et `pointeur` qui représente un pointeur abstrait vers un élément ajouté à la structure de données (ces pointeurs seront utilisés dans les fonctions `ajouter` et `rendre_plus_prioritaire` définies ci-dessous). La fonction `creer : int -> 'a t` permet de créer une file supportant un nombre n de niveaux de priorités, les priorités des éléments allant de 0 (plus prioritaire) à $n - 1$ (moins prioritaire). La fonction `ajouter : 'a -> int -> 'a t -> pointeur` permet d'ajouter un élément dans la file avec une certaine priorité. À niveau de priorité égal, c'est l'élément ajouté en dernier qui est le plus prioritaire. La valeur de type `pointeur` retournée par cette fonction doit être utilisée dans la fonction `rendre_plus_prioritaire : 'a -> pointeur -> int -> 'a t -> unit` pour changer la priorité d'un élément ajouté précédemment dans la file avec la fonction `ajouter`. Le changement de priorité ne sera effectif que si l'élément devient plus prioritaire (si ce n'est pas le cas ce changement n'aura aucun effet). Enfin, la fonction `extraire_le_plus_prioritaire : 'a t -> 'a` permet de sortir l'élément le plus prioritaire de la file.

Exemple. En supposant qu'il existe un module `F` réalisant la signature `FILE`, voici une suite d'expressions qui permet de :

- créer une file `f` à changement de priorité avec 5 priorités au plus ;
- ajouter 3 éléments à `f` ('a', 'b' et 'c') de priorité respective 4, 0 et 3 ;
- extraire l'élément le plus prioritaire de `f` ;
- changer la priorité de l'élément 'a' à 2 ;
- extraire l'élément le plus prioritaire de `f`.

```
# let f = F.creer 5;;
val f : '_a F.t = <abstr>
#let pa = F.ajouter 'a' 4 f;;
val pa : F.pointeur = <abstr>
#let pb = F.ajouter 'b' 0 f;;
val pb : F.pointeur = <abstr>
#let pc = F.ajouter 'c' 3 f;;
val pc : F.pointeur = <abstr>
#F.extraire_le_plus_prioritaire f;;
- : char = 'b'
#F.rendre_plus_prioritaire 'a' pa 2 f;;
- : unit = ()
#F.extraire_le_plus_prioritaire f;;
- : char = 'a'
```

Afin de séparer la gestion de la structure de données représentant la file à priorités du mécanisme permettant le changement de priorité, nous allons définir les files à changement de priorité comme un foncteur paramétré par un module de signature `TAS` définie de la manière suivante :

```
module type TAS = sig
  type 'a t
  val create : int -> 'a t
  val add : 'a -> int -> 'a t -> unit
  val extract_min : 'a t -> 'a
end
```

Cette signature définit un type abstrait `'a t` pour les files à priorités (appelées également *tas* dans le jargon informatique). La fonction `create : int -> 'a t` permet de créer une file vide supportant un nombre fixe n de priorités (de 0 à $n-1$, où 0 est la priorité la plus forte). La fonction `add : 'a -> int -> 'a t -> unit` est utilisée pour ajouter un élément de type `'a` avec une certaine priorité dans une file. Enfin, `extract_min : 'a t -> 'a` permet d'extraire d'une file l'élément ayant la priorité la plus forte.

La définition du foncteur `File` des file à changement de priorité est donc défini de la manière suivante :

```
module File ( T : TAS ) : FILE = struct
  ...
end
```

Afin de réaliser le mécanisme de changement de priorité, nous allons associer à chaque élément de la file une référence booléenne indiquant si cet élément est réellement présent

dans la file. Ainsi, le changement de priorité d'un élément sera simplement réalisé par l'ajout de ce même élément (avec sa *nouvelle* priorité mais la *même* référence booléenne) dans la file, les anciennes occurrences de cet élément (avec leurs anciens niveaux de priorité) étant elles toujours présentes dans la file. L'extraction de l'élément le plus prioritaire devra changer la valeur de la référence booléenne afin d'indiquer que les autres occurrences de cet élément (avec des niveaux de priorités plus élevés) ne doivent plus être considérées comme appartenant à la file. Les définitions des types `pointeur` et `'a t` du foncteur `File` sont donc les suivantes :

```
type pointeur = bool ref
type 'a t = ('a * pointeur) T.t
```

Les deux questions suivantes sont indépendantes.

Question 3.1 En utilisant les types `pointeur` et `'a t` définis ci-dessus, écrire les fonctions du foncteur `File` (chacune de ces fonctions peut s'écrire en 2 lignes au plus d'`Ocaml`).

Question 3.2 En utilisant un tableau de listes polymorphes (`'a list`) `array` pour représenter le type `'a t`, définir une implémentation d'un module `T` réalisant la signature `TAS` (la fonction la plus longue ne fait pas plus de 8 lignes d'`Ocaml`).

Rappels.

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

`List.map f [a1; ...; an]` est `[f a1; ...; f an]`

```
Array.make : int -> 'a -> 'a array
```

`Array.make n v` retourne un nouveau tableau de longueur `n`, où chaque élément est initialisé avec la valeur `v`.

```
Array.length : 'a array -> int
```

`Array.length t` retourne la longueur du tableau `t`.