Licence STS, semestre 4 — 2009–2010
Langages et Génie Logiciel (Info 223) — 16 avril 2010
`http://www.lri.fr/~paulin/LGL`

Révisions

Les notes de TD manuscrites ainsi que les fiches de cours de cette année sont les seuls documents autorisés à l'examen.

1 Typage et évaluation d'expressions Ocaml

1. Dire si les fonctions suivantes sont bien typées et si oui donner leur type :

```
let rec f a b = if !b then raise Exit else a::(f (a-1) b)
type r = { mutable p : int ref; b : float }
let f x = x.p <- ref (int_of_float x.b)
```

2. Donner les résultats de l'évaluation du programme suivant :

```
let mystere e x =
  try  if x <= 1 then raise e else x
  with | Exit -> x+1 | _ -> 0
let x = mystere Not_found 1
let y = mystere Not_found (mystere Exit 1)
```

3. Etant donné la fonction `f` suivante :

```
let f = let a = ref 1 in
  fun b -> let s = !a in a := !b; b := !a * s
```

Quelle est la valeur de `!b` à la fin de l'expression des évaluations suivantes ?

```
let b = ref 2
let _ = f b
let _ = f b
let _ = f b
```

2 Tests et preuves

Le programme suivant calcule la racine carrée entière d'un nombre entier n, c'est à dire $m \in \mathbb{N}$ tel que $m^2 \leq n < (m+1)^2$.

```
let sqrt x =
    let rec iter count sum =
    if sum <= x then iter (count+1) (sum+2*count+3) else count
    in iter 0 1
```

1. Quel est le domaine de définition de la fonction `sqrt` ?
2. Ecrire un programme de test de la fonction `sqrt` sur k entiers choisis aléatoirement entre 0 et n.
3. Ecrire un programme de test de la fonction `sqrt` sur tous les carrés i^2 pour i allant de 0 à n.
4. La fonction `iter` termine-t-elle ? pourquoi ?
5. Montrer que si $\texttt{count}^2 \leq x$ et $\texttt{sum} = (\texttt{count}+1)^2$ alors `iter count sum` calcule la racine carrée entière de x. En déduire que `sqrt` x calcule bien la racine de x.

3 Modules et foncteurs

Le but de cet exercice est de développer une structure de données efficace pour représenter des ensembles. Le type abstrait de cette structure de données est défini par la signature suivante :

```
module type SET =
sig
  type t
  type elt
  val empty : unit -> t
  val mem : t -> elt -> bool
  val add : t -> elt -> unit
  val union : t -> t -> unit
  val iter : (elt -> unit) -> t -> unit
  val choose : t -> elt
  val cardinal : t -> int
end
```

Cette signature contient deux définitions de type : le type `t` est celui des ensembles et le type `elt` celui de leur éléments. La fonction `empty` permet de créer un ensemble vide et la fonction `mem` de tester l'appartenance d'un élément à un ensemble. La fonction `add` ajoute un élément à un ensemble et `union` réalise l'union de deux ensembles. `iter` permet d'itérer une fonction sur tous les éléments d'un ensemble. La fonction `choose` renvoie un élément *quelconque* de l'ensemble passé en argument. Enfin, la fonction `cardinal` renvoie le nombre d'éléments d'un ensemble.

Le but de ce problème est d'implémenter une structure de données vérifiant la signature `SET`. Pour cela, nous allons représenter les ensembles par des tables de *hash*. Une table de *hash* permet de stocker des éléments en les regroupant par *tiroirs*, à l'aide d'une fonction de *hash* utilisée pour déterminer dans quel tiroir chaque élément doit être rangé. Si la répartition entre les différents tiroirs est équilibrée, alors chaque tiroir ne contient qu'un petit nombre d'éléments. On peut alors retrouver rapidement un élément car il ne reste plus qu'à le chercher dans son tiroir.

Concrètement, on se propose de représenter les tables de *hash* par le type Ocaml suivant :

```
type t = { data : (elt list) array ; mutable size : int }
```

où le champ `data` est un tableau de tiroirs, et chaque tiroir est une liste **sans doublons** d'éléments. Le champ `size` est utilisé pour stocker le nombre d'éléments dans la table. Chaque tiroir est donc caractérisé par son indice dans le tableau, c'est-à-dire entre 0 et la taille du tableau moins 1. La fonction de *hash* associe un entier quelconque à chaque élément et on utilise la fonction *modulo* pour déterminer le tiroir associé à cet entier.

La structure de données ne peut pas être polymorphe vis-à-vis du type des éléments car la fonction de *hash* est propre à chaque type d'élément. Nous allons donc l'implémenter comme un foncteur `HSet` paramétré par un module vérifiant la signature suivante :

```
module type HASH =
sig
  type t
  val hash : t  -> int
  val equal : t -> t -> bool
end
```

Cette signature décrit les types `t` munis d'une fonction de *hash* et également d'une opération d'égalité. Il est **important** que la fonction `equal` ait la propriété que deux valeurs égales ont la même valeur de *hash*.

La définition du foncteur `HSet` prend donc la forme suivante, où l'on introduit par ailleurs une constante `n` (fixée ici à 17) décrivant le nombre de tiroirs dont dispose la table.

```
module HSet (X : HASH) : SET = struct
  type elt = X.t
  type t = { data : (elt list) array ; mutable size : int }

  let n = 17
  ...
end
```

1. À partir de la signature `SET`, que pouvez-vous dire sur l'aspect fonctionnel (ou persistant) de la structure que l'on cherche à implanter ?
2. Écrire les fonctions `empty : unit -> t` et `cardinal : t -> int`.
3. Écrire la fonction `mem : t -> elt -> bool` telle que `mem s x` renvoit `true` si et seulement si `s` contient un élément égal à `x` au sens de l'égalité apportée par le module `X`.
4. En utilisant éventuellement la fonction `mem` de la question précédente, écrire la fonction d'ajout d'un élément `add : t -> elt -> unit`.
5. Écrire une fonction `merge_list : list elt -> list elt -> list elt` telle qu'étant donné deux listes sans doublons `l1` et `l2`, `merge_list l1 l2` renvoit une liste également sans doublons et contenant tous les éléments présents dans `l1` ou `l2`. On s'aidera alors de cette fonction afin d'écrire la fonction `union : t -> t -> unit` qui calcule l'union de deux ensembles avec la contrainte suivante : `union s1 s2` doit laisser `s1` inchangé et calculer l'union de `s1` et `s2` dans `s2`.
6. Écrire la fonction d'itération `iter : (elt -> unit) -> t -> unit` qui permet d'itérer une fonction sur tous les éléments d'un ensemble. L'ordre dans lequel les éléments doivent être parcourus n'est pas spécifié.
7. Écrire la fonction `choose : t -> elt` qui renvoit un élément quelconque d'un ensemble. Si aucun élément n'est trouvé, *i.e.* si l'ensemble est vide, la fonction lèvera l'exception `Not_found`.
   Indice : on pourra définir une exception `Found of elt` *et utiliser l'itérateur réalisé à la question précédente.*
8. On a maintenant terminé l'implémentation du foncteur `HSet` et on souhaite l'utiliser sur des types d'éléments particuliers. Écrire un module `BFHash` de signature `HASH` pour le type `t = bool * float` :

   ```
   module BFHash = struct
     type t = bool * float
     ...
   end
   ```

   Instancier alors le foncteur `HSet` sur ce module afin d'obtenir un module `BFSet` d'ensembles de paires booléen-flottant.
9. On veut utiliser le module `BFSet` pour faire les manipulations suivantes mais l'ajout est refusé par le typage d'OCaml :

   ```
   # let s = BFSet.empty ();;
   val s : BFSet.t = <abstr>
   # BFSet.add s (bool, 1.5);;
   This expression has type bool * float but is here used with type BFSet.elt
   ```

   Expliquer l'origine de l'erreur rencontrée. Quelle contrainte faut-il ajouter à la signature du foncteur `HSet` pour corriger ce problème ?

10. Est-il possible d'utiliser `HSet` pour construire facilement des ensembles d'ensembles d'un type donné ? Si oui, décrivez comment ; dans le cas contraire, que faudrait-il rajouter au foncteur `HSet` ?

4 Analyse lexicale

Ecrire un programme `ocamllex` qui reconnait dans un fichier les chaînes de caractères qui peuvent correspondre à des adresses électroniques et les imprime.

- Une adresse électronique sera de la forme *identifiant*`@`*machine*`.`*dom*
- L'identifiant et le nom de machine peuvent contenir des caractères alphabétiques, numériques et des caractères spéciaux comme `.` `_` et `-`.
- Le domaine est formé de deux ou trois caractères alphabétiques.
- Le programme prendra en argument un nom de fichier, il imprimera sur la sortie standard les adresses identifiées et ignorera les autres éléments.