Université Paris Diderot
M1 Info

Programmation Fonctionnelle Avancée
Année 2016-2017

Examen

Jeudi, 12 janvier 2017

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints et rangés. Le temps à disposition est de 2.5 heures.

Les exercices doivent être rédigés en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Chaque fonction ci-dessous peut utiliser les fonctions prédéfinies (sauf indication contraire), et/ou les fonctions des questions précédentes.

Cet énoncé a 4 pages.

Exercice 1 L'objectif de cet exercice est d'implémenter un module d'ensembles de valeurs d'un type ordonné, basés sur les intervalles.

1. Soit le code suivant :

```
module type ORDER = sig
  type t
  val leq : t -> t -> bool
  equal : t -> t -> bool
end

module Int:ORDER with type t = int = struct
  type t = int
  let leq = (<=)
  let equal = (=)
end
```

   Expliquer brièvement (quelques lignes) à quoi sert le `with type t=int` dans la définition du module `Int`.

2. Soit la signature suivante :

```
module type INTERVAL = sig
  exception NoOverlap
  type element
  type t
  val create: element -> element -> t
  val equal: t -> t -> bool
  val mem: element -> t -> bool
  val before: t -> t -> bool
  val intersection: t -> t -> t
  val union: t -> t -> t
end
```

Définir un foncteur `MakeInterval` qui prend en paramètre un module de signature `ORDER`, et qui renvoie un module de signature `INTERVAL`. Dans le module envoyé, le type `t` réalise les intervalles fermés de valeurs de type `element`. La fonction `create` crée un tel intervalle pour une borne inférieure et une borne supérieure données, `equal` teste si deux intervalles sont égaux, `mem` teste si une valeur appartient à un intervalle, `before` teste si un premier intervalle est strictement devant un deuxième intervalle (on suppose que l'ordre des valeurs est strict, c'est-à-dire que $y > x$ est équivalent à $y \not\leq x$). Les fonctions `intersection` et `union` calculent l'intersection, resp. l'union de deux intervalles dans le cas où ils contiennent au moins un élément commun, et lèvent l'exception `NoOverlap` sinon.

Par exemple, le code suivant devra répondre `false` :

```
module I=MakeInterval(Int)
I.mem 42 (I.intersection (I.create 51 81) (I.create 40 60))
```

Indication : la fonction `before` peut aider à tester si deux intervalles ont un élément commun. Vous avez évidement le droit de définir des fonctions privées qui ne seront pas exportées par le module.

3. Soit la signature suivante :

```
module type SET = sig
  type element
  type t
  val interval: element -> element -> t
  val union: t -> t -> t
  val mem: element -> t-> bool
  val equal: t -> t -> bool
end
```

Écrire un foncteur `MakeSet` qui prend en paramètre un module de signature `ORDER`, et renvoie un module de signature `SET`. En général, un ensemble `t` de ce module va être une union de plusieurs intervalles disjoints. Il convient de représenter une telle union d'intervalles par une liste ordonnée d'intervalles. La fonction `union` doit donc respecter cette représentation. Pour le codage de chaque intervalle, vous devez vous servir du foncteur `MakeInterval` de la question précédente. La fonction `interval` crée un ensemble constitué d'un seul intervalle. La fonction `mem` teste l'appartenance à un ensemble, et `equal` teste l'égalité de deux ensembles.

4. On imagine maintenant la signature `SET` étendue par

```
val intersection : t -> t -> bool
```

Donner une implémentation de cette fonction `intersection` pour le foncteur `MakeSet`.

Exercice 2 Cet exercice présente une méthode de calcul de la liste infinie des puissances d'entiers : $1^n, 2^n, 3^n, 4^n, ...$ en utilisant uniquement des additions. Pour représenter les listes infinies, nous utiliserons ici le type `stream` suivant :

```
type 'a stream = 'a cell Lazy.t
and 'a cell = Cons of 'a * 'a stream
```

Voici par exemple une fonction donnant la liste des `n` premiers éléments d'une telle stream.

```
let rec take n s =
  if n = 0 then []
  else match s with lazy (Cons (x,s')) -> x :: take (n-1) s'
```

1. Construire la stream `numbers` contenant tous les entiers naturels non-nuls : $1, 2, 3, 4, ...$
2. Écrire une fonction `sum : int stream -> int stream` calculant la stream des *sommes partielles* des premiers éléments de l'entrée. Par exemple, avec la stream `numbers` définie à la question précédente, `sum numbers` doit correspondre à $1, 1+2, 1+2+3, 1+2+3+4, ...$ autrement dit à $1, 3, 6, 10, ...$
3. Écrire une fonction `decimate : int -> 'a stream -> 'a stream` telle que `decimate k s` est bâtit sur le principe suivant : on prend `k` éléments dans `s`, on jète l'élément suivant, et on recommence. Par exemple, `decimate 2 numbers` correspond à $1, 2, 4, 5, 7, ...$
4. Écrire une fonction `moessner : int -> int stream` de sorte que `moessner n` calcule $1^n, 2^n, 3^n, 4^n, ...$ en utilisant l'algorithme suivant (dû à Moessner) : on part de `numbers`, sur lequel on fait un `decimate (n-1)` puis un `sum`, puis un `decimate (n-2)`, puis de nouveau `sum`, et ainsi de suite jusqu'à `decimate 1` et un dernier `sum`. La stream finale est alors celle recherchée.
5. Lors du calcul de `take 3 (moessner 4)`, combien de cellules de la stream `number` sont-elles dégelées ? Et combien d'additions sont-elles réalisées ? On suppose ici que `numbers` vient juste d'être fraîchement définie.

Exercice 3 Dans cet exercice nous allons utiliser les GADT pour représenter des séquences polymorphes, et aussi pour représenter les arbres binaires complets polymorphes. Dans les deux cas, un premier paramètre de type indiquera le type des valeurs stockées dans les séquences, resp. arbres ; un deuxième paramètre de type indiquera la longueur d'une séquence, resp. la hauteur d'un arbre.

1. Soit la définition suivante d'un GADT :

   ```
   type ('a,'l) sequence =
     | Nil: ('a,unit) sequence
     | Cons: 'a * ('a,'l) sequence -> ('a,'l*unit) sequence
   ```

   Ce type permet de représenter des séquences dont les éléments sont de type `'a`. Le paramètre de type `'l` indique au niveau du typage la longueur d'une séquence. Donner les types des expressions suivantes :

   ```
   Nil
   Cons(1,Cons(2,Nil))
   ```

2. Un *arbre complet de hauteur n* est défini comme suit :
   — `Leaf` est un arbre complet de hauteur 0 ;

— `Noeud(tl,v,tr)` est un arbre complet de hauteur $n+1$ quand `tl` et `tr` sont des arbres complets de hauteur n.

Voici un canvas d'une définition d'un GADT pour les arbres complets :

```
type ('a,'h) bintree =
  | Leaf: ('a,??) bintree
  | Node: ('a,??) bintree * 'a * ('a,??) bintree -> ('a,??) bintree
```

Compléter ce canvas en remplaçant les `??` par des bonnes expressions. L'idée est que le type `bintree` permettra de représenter seulement des arbres complets, mais pas des arbres qui ne sont pas complets. Par exemple, après

```
let tx = Node (Node (Leaf, 1, Leaf), 3, Node (Leaf, 2, Leaf))
```

le type de `tx` est `(int, (unit * unit) * unit) bintree`.

3. Écrire une fonction `treemap` qui prend en paramètre une fonction f et un arbre complet t (de type `bintree`), et qui envoie un arbre complet de la même hauteur que t_1, et dont les valeurs dans les nœuds sont obtenues par application de f aux valeurs correspondantes de t, par exemple, `treemap (function x -> x+1) tx` doit donner l'arbre

   ```
   Node (Node (Leaf, 2, Leaf), 4, Node (Leaf, 3, Leaf))
   ```

4. Écrire une fonction `treemap2` qui prend en paramètre une fonction f et deux arbres complets t_1 et t_2 (de type `bintree` et de la même hauteur), et qui envoie un arbre complet de la même hauteur que t_1 et t_2, et dont les valeurs dans les nœuds sont obtenues par application de f aux valeurs correspondantes de t_1 et t_2, par exemple, `treemap2 (fun x y -> x*y) tx tx` doit donner l'arbre

   ```
   Node (Node (Leaf, 1, Leaf), 9, Node (Leaf, 4, Leaf))
   ```

5. Écrire une fonction `shrink` qui prend en paramètre une fonction f à deux arguments et un arbre complet t (de type `bintree`), et qui envoie une séquence de la même longueur que la hauteur de t, et dont le i-ième élément est obtenu par une combinaison de toutes les valeurs au i-ième niveau de t. Vous êtes libre de choisir l'ordre dans lequel ces valeurs sont combinées. Par exemple, après

   ```
   let ty=Node(
               Node(Node(Leaf,4,Leaf), 2, (Node(Leaf,5,Leaf))),
               1,
               Node(Node(Leaf,6,Leaf), 3, (Node(Leaf,7,Leaf))))
   ```

   le résultat de `shrink (+) ty` doit être `Cons(1,Cons(5,Cons(22,Nil)))`.

   Indication : Servez-vous de la fonction `treemap2`. Le résultat de `shrink f t` est la séquence qui commence sur la valeur à la racine de t. Pour obtenir le reste de la séquence, on peut combiner les deux sous-arbres de t à l'aide de `treemap2`, et puis continuer récursivement.

6. Écrire, en utilisant seulement les fonctions `treemap` et `shrink`, une fonction `collect_levels` qui prend en paramètre un arbre complet t, et qui envoie une séquence de la même longueur que la hauteur de l'arbre t, et dont le i-ième élémemt est la liste de toutes les valeurs de t au i-ème niveau, dans n'importe quel ordre. Par exemple, le résultat de `collect_levels ty` pourra être

   ```
   Cons([1],Cons([2;3],Cons([4;6;5;7],Nil)))
   ```