Université Paris Cité
M1 Info

Programmation Fonctionnelle Avancée
Année 2021-2022

Examen

16 mai 2022

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.

1 Compte à 100 et découpe de listes

Rappel :

- `List.concat [[1;2];[];[3]] = [1;2;3]`
- `String.concat "," ["a";"b";"c"] = "a,b,c"` .

Exercice 1 On cherche à résoudre en OCaml la devinette suivante :

> *Prendre les nombres de 1 à 9 dans cet ordre, et mettre entre chaque soit un + soit un * de telle manière que le résultat du calcul soit 100.*

On ne peut pas utiliser de parenthèses, et comme d'habitude une multiplication est plus prioritaire qu'une addition. Par exemple, une solution possible est `1*2*3+4+5+6+7+8*9`. On choisit ici de représenter ce genre d'expression par le type OCaml `int list list`, la liste extérieure représentant les additions et les listes intérieures représentant les multiplications. Ainsi l'expression solution précédente est codée `[[1;2;3];[4];[5];[6];[7];[8;9]]`.

1.1 Écrire une fonction `printexpr : int list list -> string` qui transforme le codage précédent en la chaîne de caractères associée (ici `"1*2*3+4+5+6+7+8*9"`).

1.2 Écrire une fonction `evalexpr : int list list -> int` qui calcule la valeur de l'expression (ici 100).

Un *découpage* d'une liste `l` est une liste de listes `ll` dont tous les éléments sont des listes non-vides, et telle que la concaténation de tous ces éléments `List.concat ll` redonne la liste de départ `l`. Par exemple `[[1;2];[3]]` et `[[1];[2];[3]]` sont deux découpages possibles de `[1;2;3]`, et il y en a deux autres.

1.3 Écrire une fonction `splits : 'a list -> 'a list list list` qui produit la liste de tous les découpages possibles de la liste donnée en entrée. Ces découpages peuvent être listés dans l'ordre de votre choix.

1.4 Utiliser cette fonction `splits` pour générer la liste de toutes les expressions que l'on peut obtenir en insérant des + et des * entre les nombres de 1 à 9, puis la liste des expressions donnant 100 comme résultat. On ne demande pas d'effectuer ce calcul !

2 Intervalles

Exercice 2 L'objectif de cet exercice est d'implémenter un module d'ensembles de valeurs d'un type ordonné, basés sur les intervalles.

2.1 Soit le code suivant :

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

2.2 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 total, 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.

2.3 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.

2.4 On imagine maintenant la signature `SET` étendue par

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

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

2.5 Indiquer succinctement ce qu'il faudrait modifier au code précédent pour pouvoir accepter également les intervalles infinis, c'est-à-dire ayant $-\infty$ pour borne inférieure et/ou $+\infty$ pour borne supérieure ?

3 Flot des puissances

Exercice 3 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'
```

3.1 Construire la stream `numbers` contenant tous les entiers naturels non-nuls : $1, 2, 3, 4, ...$

3.2 Écrire une fonction `sum : int stream -> int stream` donnant 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 $1, 3, 6, 10, ...$

3.3 Écrire une fonction `decimate : int -> 'a stream -> 'a stream` enlevant régulièrement certains éléments de la stream d'entrée. Plus précisément, `decimate k s` est bâti sur le principe suivant : on prend `k` éléments dans `s`, on jette l'élément suivant, et on recommence. Par exemple, `decimate 2 numbers` correspond à $1, 2, 4, 5, 7, ...$

3.4 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.

3.5 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.