Université Paris Diderot
M1 Info

Programmation Fonctionnelle Avancée
Année 2015-2016

Examen

Vendredi, 15 janvier 2016

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 3 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 5 pages.

Exercice 1 La figure 1 contient une implémentation naïve d'une base de données qui associe à des clefs du type `string` des valeurs de type `int`. Le type `request` définit différentes requêtes : chercher la valeur associée à une clef, supprimer une clef, ou ajouter une paire clef-valeur. La fonction `exec` exécute une requête sur une base de donnée. Remarquez que le type du résultat dépend de la requête ; le type somme `result` permet de représenter des résultats qui peuvent être une base de données ou un string.

1. Donner un terme, utilisant les fonctions `exec`, `extractdb`, `extractint`, qui représente l'enchaînement des trois requêtes suivantes :
   ajouter `("a",1)` à une base de données vide
   puis ajouter `("b",2)`
   puis extraire la valeur associée à la clef `"a"`
   Le résultat de l'évaluation de ce terme doit donc être 1.
2. On cherche maintenant une meilleure implémentation utilisant les GADT. Donner une nouvelle définition du type `request` en tant que GADT, tel que `Get` construit une valeur de type `int request`, et tel que `Ins` et `Del` construisent des valeurs de type `db request`.
3. Modifier la fonction `exec` pour le nouveau type `request` obtenu à la question précédente. Quel est son type ?
4. Même question que (1), mais maintenant pour le type `request` obtenu à la question (2) et la fonction `exec` obtenue à la question (3).

Exercice 2

1. Définir une signature du nom `ADD` qui contient un type `t`, et une fonction `add` qui prend deux arguments du type `t` et retourne une valeur du même type. Puis, définir une deuxième signature `ADDMULT` qui étend, en utilisant la construction `include`, la signature `ADD` par une constante `null` de type `t`, et une fonction `mult` qui prend deux arguments du type `t` et retourne une valeur du même type.
2. Définir un module `Int` de signature `ADDMULT` dont le type `t` consiste en les entiers, `null` est 0, et `add` et `mult` sont les opérations arithmétiques habituelles.

```
type entry = string * int;;

type db = entry list;;

let empty:db = [];;

type request =
  | Ins of entry
  | Del of string
  | Get of string
;;

type result = Db of db | Found of int;;

let exec x = function
  | Get s -> Found (List.assoc s x)
  | Del s -> Db (List.remove_assoc s x)
  | Ins e -> Db (e::x)
;;

let extractdb = function
  | Db x -> x
  | _ -> failwith "not a data base"
;;

let extractint = function
  | Found i -> i
  | _ -> failwith "not an integer"
```

FIGURE 1 Implémentation naïve d'une base de données

3. Définir un foncteur `AddVector` qui prend en argument une structure de signature `ADD`. Ce foncteur retourne une structure contenant :
   - un type `t` qui correspond aux listes dont les éléments sont du type `t` de la structure paramètre ;
   - une fonction `add` qui prend deux listes (supposées de même taille), et qui retourne la liste des sommes des éléments des deux listes (en utilisant la fonction `add` de la structure paramètre) ;
   - une fonction `map`, analogue à la fonction `map` des listes
4. Définir un foncteur `Vecteur` qui associe à une structure paramètre de signature `ADDMULT`, une structure qui étend la structure retournée par `AddVector` par les deux fonctions suivantes :
   - une fonction binaire `inner_product` qui calcule le produit scalaire de deux vecteurs (en utilisant les fonctions `add` et `mult` de la structure paramètre) ;
   - une fonction binaire `multiple` qui retourne le produit entre une constante et un vecteur (tous les éléments de ce vecteur sont multipliés par la même constante).

```
let rec times n = function
| lazy Nil -> lazy Nil
| lazy (Cons(h,t)) -> lazy (Cons(n*h,times n t))
```

FIGURE 2 Version 1 de la fonction `times`

```
let times n s =
  let rec times_aux = function
   | lazy Nil -> Nil
   | lazy (Cons(h,t)) -> (Cons(n*h,lazy (times_aux t)))
  in lazy (times_aux s);;
```

FIGURE 3 Version 2 de la fonction `times`

5. Définir un foncteur `Matrix` qui à une structure de signature `ADDMULT` associe une structure de matrices dont les éléments sont des valeurs de la structure paramètre. Il n'est pas demandé de vérifier que la forme de la matrice est rectangulaire : on autorise des matrices dont les lignes ont des longueurs différentes. La structure doit contenir les deux fonctions suivantes :
   une fonction binaire `add` qui calcule la somme de deux matrices (en supposant que les deux matrices ont les mêmes dimensions) ;
   une fonction binaire `multiple` qui retourne le produit entre une constante et une matrice (tous les éléments sont multipliés avec la même constante).
   La structure des matrices doit impérativement être construite à l'aide du foncteur `Vector` ; les définitions de fonctions dans ce foncteur doivent au maximum utiliser des fonctions fournies par le foncteur `Vector`.

Exercice 3 Soit la définition suivante d'un type de flots :

```
type 'a streamtip = Nil | Cons of 'a * 'a stream
and  'a stream = 'a streamtip Lazy.t;;
```

1. Les figures 2 et 3 contiennent deux versions différentes d'une fonction `times` qui prend en paramètre un entier n et un flot d'entiers $i_0; i_1; i_2; \ldots$, et qui retourne comme résultat le flot $n*i_0; n*i_1; n*i_2; \ldots$. Est-ce que les deux versions de la fonction ont exactement le même comportement ? Justifiez ! (5 à 10 lignes maximum)
2. Écrire une fonction `merge` qui prend en paramètre deux flots d'entiers s_1 et s_2 qui sont supposés être triés dans un ordre strictement ascendant (en particulier, sans doublons), et qui retourne le flot de tous les entiers qui apparaissent dans s_1 ou s_2, également dans un ordre strictement ascendant.
3. La *séquence de Hamming particulière* est le flot de tous les entiers de la forme $2^i * 3^j * 5^k$ pour $i, j, k \geq 0$, dans l'ordre strictement ascendant. Le début du flot est $1; 2; 3; 4; 5; 6; 8; 9; 10; 12; 15; 16; 18; \ldots$. Définir ce flot en OCaml.
4. Étant donnée une liste d'entiers $l = [p_0; \ldots; p_n]$, la *séquence de Hamming avec facteurs* l est le flot de tous les entiers strictement positifs dont tous les facteurs premiers sont des éléments de l. La séquence de Hamming particulière de la question précédente est

```
type exp =
    | Int of int
    | Iden of string
    | Plus of (exp * exp)
    | App of (exp * exp)
    | Abs of (string * exp)

module type Monad = sig
    type 'a t
    val return : 'a -> 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
  end

module Interp (M: Monad) =
struct
 type idt = VInt of int | VArrow of (idt -> expt)
 and  expt = idt M.t
 let ( >>= ) = M.bind
 let return = M.return
 let rec interp (exp:exp) (env: (string * idt) list) : expt =
   match exp with
   | App (e1,e2) -> interp e1 env >>= fun fv ->
            (match fv with
            | VArrow f -> interp e2 env >>= fun rv -> f rv
            | _ -> failwith "Application of non function")
   | Abs (string,exp)->
       return
         (VArrow(fun a -> (interp exp ((string,a)::env))))
   | Iden(id)     -> return (List.assoc id env)
   | Int i        -> return (VInt i)
   | Plus(e1,e2)  ->
         interp e1 env >>= fun v1 ->
         interp e2 env >>= fun v2 ->
         return (match (v1,v2) with
                 | (VInt x1, VInt x2) -> VInt (x1+x2)
                 | _ -> failwith "Not an Integer")
end
```

FIGURE 4 Un interpréteur monadique simplifié

donc la séquence de Hamming avec facteurs $[2; 3; 5]$. Définir une fonction en OCaml qui prend en argument une liste d'entiers l, et qui retourne la séquence de Hamming avec facteurs l.

Exercice 4 La figure 4 contient une version simplifiée de l'interpréteur monadique donné en cours. Les simplifications concernent le type des Booléens, les opérateurs de base, et l'utilisa-

tion de types algébriques au lieu de variants polymorphes.

1. Combien d'appels à `bind` seront exécutés par l'interprétation du terme suivant ? Il suffit de donner le nombre.

```
(Plus(
      (App(Abs("x",Iden "x"),Iden "x0")),
      (App(Abs("y",Plus(Iden "y",Iden "y")), Int 5))
   )
 );;
```

2. Donner une implémentation d'un module `CM` de signature `Monad` tel qu'un appel à `Interp(CM).inter exp env` retourne la valeur obtenue par l'interprétation de l'expression `exp` dans l'environnement `env`, et en plus le nombre d'appels à `bind`. Par exemple, invoquée sur le terme de la question précédente et l'environnement `[("x0",Int 1)]`, la fonction doit retourner $(n, 11)$ où n est la réponse à la question (1). Pour cette question vous avez seulement le droit de définir la structure `CM`, aucune modification du module `Interp` n'est autorisée
3. Est-ce que la structure `CM` satisfait les axiomes des monades vues en cours ? Justifiez.
4. Considérons maintenant le module `SM` de signature `Monad` suivant :

```
module SM = struct
  type 'a t = int -> ('a * int)
  let return v = function i -> (v,i)
  let bind a f = function n -> let (w,j) = a n in f w j
end
```

Montrer que ce module satisfait l'équation suivante pour tout entier n, et pour tout m d'un type `'a t` :

`bind m return n = m n`