Master d'Informatique
Année 2012-2013
Examen de Programmation Fonctionnelle Avancée

Tous documents autorisés

Durée: 3h

Première Session: 13 Mai 2012

Zippers (6 points)

Exercice 1 *Visite d'une structure à l'aide d'un zipper*

Nous souhaitons utiliser des zippers pour naviguer dans l'arbre de syntaxe abstraite d'un petit langage d'expressions, décrit par le type de données suivant:

```
type expr =
  | Var of string
  | Entier of int
  | Plus of expr * expr
  | Mul  of expr * expr;;
```

On vous demande de

1. *dériver le type d'un zipper pour cette structure;*
2. *écrivez les fonctions permettant de se déplacer dans la structure: en haut, en bas à droite, en bas à gauche*

Solution 1 *Il faut dériver le type du bloc de pile en suivant la technique vue en cours, introduite par Conor McBride.*
Cela donne: $F = 1+1+(x*x)+(x*x)$, *dont la dérivée est* $F' = (x+x)+(x+x)$, *et en remplaçant, on obtient le type* `(expr+expr)+(expr+expr)`, *qu'on peut encoder en OCaml de plusieurs façons; ci-de suite en voici une:*

```
type direction = Gauche | Droite;;
type op = BPlus | BMul;;

type bloc =  op * direction * expr;;

type pile = bloc list;;

type z_expr = pile * expr;;

exception EnBas;;
```

```
exception EnHaut;;

let bas_gauche (pile, e) : z_expr = match e with
  | Plus(x, y) -> (BPlus, Gauche, y)::pile, x
  | Mul(x, y)  -> (BMul, Gauche, y)::pile, x
  | _ -> raise EnBas;;

let bas_droite (pile, e) : z_expr = match e with
  | Plus(x, y) -> (BPlus, Droite, x)::pile, y
  | Mul(x, y)  -> (BMul, Droite, x)::pile, y
  | _ -> raise EnBas;;

let en_haut (pile, e) : z_expr = match pile with
  | (BPlus, Gauche, y)::p -> (p, Plus(e, y))
  | (BPlus, Droite, x)::p -> (p, Plus(x, e))
  | (BMul, Gauche, y)::p -> (p, Mul(e, y))
  | (BMul, Droite, x)::p -> (p, Mul(x, e))
  | _ -> raise EnHaut;;
```

Remarque Les exercices proposés dans la suite ont pour but d'écrire un solveur de Sudoku. Dans chaque exercice, vous pouvez supposer de disposer des fonctions définies dans les exercices précedents, même si vous n'avez pas réussi à les écrirer vous-mêmes.

Manipulation d'une grille (7 points) On represente les grilles d'un Sudoku nxn comme une liste de listes de cases contenant les entier 1 à n, l'entier 0 representant une case vide. Voici les définitions correspondantes, et un exemple de grille.

```
type 'a row = 'a list
type 'a matrix = 'a row list
type grid = int matrix
let easy : grid =
  [[ 1; 3; 2; 4];
   [ 2; 4; 1; 3];
   [ 3; 0; 4; 0];
   [ 4 ; 0; 3; 0]];;
```

Exercice 2 (Lignes et Colonnes) *Pour manipuler une grille nxn de Sudoku representée comme indiqué ci-dessus, nous avons besoin d'écrire les fonctions suivantes qui retournent la liste des lignes et des colonnes d'une matrice.*

```
rows: 'a matrix -> 'a matrix
cols: 'a matrix -> 'a matrix
```

La fonction `rows` *est triviale:*

```
let rows g = g;;
```

Donnez la définition de la fonction `cols`*, dont voici un exemple d'execution.*

```
let test =
[[ 1; 2; 3; 4];
 [ 5; 6; 7; 8];
 [ 9; 10; 11; 12];
 [13; 14; 15; 16]];;
cols test;;
```

```
-: int list list =
[[1; 5;  9; 13];
 [2; 6; 10; 14];
 [3; 7; 11; 15];
 [4; 8; 12; 16]]
```

La fonction `cols` effectue une *transposition* de matrices. Notons au passage que `fun m -> rows (rows m) = fun m -> m = fun m -> cols (cols m)`; aussi, si la matrice `m` en entrée est carrée, `cols m` est carrée aussi, de même taille.

Solution 2 *Il y a beaucoup de façons de repondre à cette question. La plus concise est probablement la suivante, qui utilise une exception pour capturer le cas ou `m` contient au moins une liste vide.*

```
(* cut columns one after the other *)
let rec cols = function
    [] -> []
  | m  -> (List.map List.hd m)::(try cols (List.map List.tl m) with _ -> []);;
```

On peut se passer d'exceptions, cependant, avec du code un peu moins concis:

```
let headtails ll =
  List.fold_right
   (fun (h::t) (hl,tl) -> h::hl,t::tl)
   ll ([],[]);;

let rec transpose  = function
   []                -> []
|  []    :: xss     -> transpose xss
|  (x::xs) :: xss -> let (hl, tl) = headtails xss in
     (x :: hl) :: transpose (xs :: tl);;
```

Ou encore:

```
let cols = function
    [] -> []
  | r::g ->
     List.map List.rev
       (List.fold_left
         (fun acc r' -> List.map2 (fun e r -> e::r) r' acc)
         (List.map (fun x -> [x]) r) g);;
```

Toutes ces solutions ont eu le maximum des points. Des solutions utilisant des `List.nth` ont été fournies, mais moins élégantes, et plus chères à l'éxécution.

Exercice 3 (Carrés) *Une grille nxn de Sudoku est divisée en n carrès de n cases (ce qui veut dire que n doit être un carré $n = k^2$, et on peut donc faire des grilles de Sudoku de taille 1x1, 4x4, 9x9, ... mais pas 5x5).*

En profitant du fait que un carré a la même taille qu'une ligne, nous pouvons écrire une fonction `boxes: int -> 'a matrix -> 'a matrix` qui utilise l'entier k et retourne la liste des carrés d'une grille de Sudoku, chacun representé comme la liste (dans l'ordre) des elements du carré.
Par exemple:

Pour obtenir ce résultat de façon fonctionnelle, on peut procéder via les étapes suivantes:

```
[[ 1;  2;  3;  4];        [[[ 1;  2]; [ 3;  4]];
 [ 5;  6;  7;  8];  -->    [[ 5;  6]; [ 7;  8]]; -->
 [ 9; 10; 11; 12];         [[ 9; 10]; [11; 12]];
 [13; 14; 15; 16]]         [[13; 14]; [15; 16]]]

 [[[[ 1;  2]; [ 3;  4]];        [[[[ 1;  2]; [ 5;  6]];
   [[ 5;  6]; [ 7;  8]]];  -->    [[ 3;  4]; [ 7;  8]]];  -->
  [[[ 9; 10]; [11; 12]];         [[[ 9; 10]; [13; 14]];
   [[13; 14]; [15; 16]]]]         [[11; 12]; [15; 16]]]]

[[ 1;  2;  5;  6];
 [ 3;  4;  7;  8];
 [ 9; 10; 13; 14];
 [11; 12; 15; 16]]
```

Ecrivez le code de `boxes`. On notera ici aussi que `fun m -> boxes (boxes m) = fun m -> m`

Solution 3 *Il s'agissait essentiellement d'écrire une fonction capable de découper une liste en blocs d'une longueur donnée, le reste étant assez naturel.*

```
let chop n l =
  let rec aux acc1 acc2 k l = match k,l with
  | _,[]     -> acc1::acc2
  | 0,l'     -> aux [] (acc1::acc2) n l'
  | _,a::r   -> aux (a::acc1) acc2 (k-1) r
  in List.rev (List.map List.rev (aux [] [] n l))
;;

let boxes boxsize g =
  let split g = chop boxsize g in
  let pack g = split (List.map split g) in
  let unpack g = List.map List.concat (List.concat g)
  in unpack (List.map cols (pack g))
;;
```

Resolution d'une grille (7 points) Avec les fonctions precedentes, il est facile d'ecrire une fonction `valid: int -> int matrix -> bool` qui verifie si une grille respecte les regles du Sudoku; nous supposons cette fonction écrite.

Solution 4 `let forall p l = List.fold_left (fun acc x -> (p x) && acc) true l;;`

```
let rec alldifferent = function
    [] -> true
  | a::r -> not (List.mem a r) && alldifferent r;;

let valid boxsize g =
  forall alldifferent (rows g) &&
  forall alldifferent (cols g) &&
  forall alldifferent (boxes boxsize g)
;;
```

Exercice 4 (Explicitation des choix) *Etant donnée une grille, nous pouvons la transformer pour rendre explicites tous les choix qui restent à faire pour resoudre le puzzle: on remplacera*

chaque occurrence de 0 *par la liste de tous les valeurs possibles (de* 1 *à* n*), et tout autre entier* m *par la liste* $[m]$*.*

Ecrivez une fonction `choices: int -> int matrix -> int list matrix` *qui réalise cette opération.*

Voici un exemple:

```
choices 2 easy;;
- : int list matrix =
[[[1]; [3]; [2]; [4]];
 [[2]; [4]; [1]; [3]];
 [[3]; [1; 2; 3; 4]; [4]; [1; 2; 3; 4]];
 [[4]; [1; 2; 3; 4]; [3]; [1; 2; 3; 4]]]
```

Solution 5 *Cette partie est assez élémentaire, et heureusement la plupart d'entre vous l'a codée correctement.*

```
let empty c = c = 0;;

let range i j =
  let rec aux acc n = if n=i then i::acc else aux (n::acc) (n-1)
  in aux [] j;;

let choices n g =
   List.map
     (List.map (fun v -> if empty v then range 1 (n*n) else [v]))
   g
;;
```

Exercice 5 (Simplification) *Les regles du Sudoku nous disent que s'il y a dans la meme ligne, colonne, ou carre d'une case donnée une liste* `[m]` *contenant un seul entier* m*, alors il est inutile de garder cet* `m` *parmi les choix de cette case.*

Ecrivez une fonction `simplify: int -> int list matrix -> int list matrix` *qui réalise la simplification par lignes, colonnes et carrés.*

Voici un exemple:

```
simplify 2 (choices 2 easy);;
- : int list list list =
[[[1]; [3]; [2]; [4]];
 [[2]; [4]; [1]; [3]];
 [[3]; [1; 2]; [4]; [1; 2]];
 [[4]; [1; 2]; [3]; [1; 2]]]
```

Vous pouvez profiter du fait que les fonctions `rows`, `cols` et `boxes` sont involutives pour écrire du code très concis, et il peut être utile d'utiliser la compréhension des listes ou la monade des listes.

Solution 6 *Il y a évidemment plusieurs façons de traiter cette question. Ici on utilise une solution due à Richard Bird.*

```
(* List Monad *)
module ListMonad =
  struct
    type 'a t = 'a list
```

```
    let return x = [x]
    let bind l f = List.fold_right (fun x acc -> (f x)@acc) l []
    let zero = []
    let ( >>= ) l f  = bind l f
  end;;

open ListMonad;;

(* infix composition notation *)
let ( |> ) x f = f x;;

let ldiff l l' = List.filter (fun x -> not (List.mem x l')) l;;

let single = function [_] -> true | _ -> false;;

let minus r r' = if single r then r else ldiff r r';;

let restrict rl =
  let singles = List.flatten (List.filter single rl) in
  rl >>= (fun xs -> return (minus xs singles));;

let simplifyBy f g =
    g |> f
      |> List.map restrict
      |> f;;

let simplify boxsize g =
   g |> simplifyBy rows
     |> simplifyBy cols
     |> simplifyBy (boxes boxsize);;
```

Exercice 6 (Expansion) *En utilisant la comprehension des listes, ou la monade des listes, écrivez une fonction* `expand: 'a list list -> 'a list list` *qui prend une liste contenant dans chaque position une liste des valeurs admis, et produit la liste des listes obtenues en prenant de toutes les facons possibles un seul des valeurs admis.*

Par exemple

```
expand [[4]; [1; 2; 3; 4]; [3]; [1; 2; 3; 4]];;
- : int list list =
[[4; 1; 3; 1]; [4; 1; 3; 2]; [4; 1; 3; 3]; [4; 1; 3; 4]; [4; 2; 3; 1];
 [4; 2; 3; 2]; [4; 2; 3; 3]; [4; 2; 3; 4]; [4; 3; 3; 1]; [4; 3; 3; 2];
 [4; 3; 3; 3]; [4; 3; 3; 4]; [4; 4; 3; 1]; [4; 4; 3; 2]; [4; 4; 3; 3];
 [4; 4; 3; 4]]
```

Solution 7 *Pour cette question, l'utilisation de la comprehension des listes, ou de la monade des listes, est cruciale pour écrire du code simple et élegant.*

```
let rec expand = function
    [] -> [[]]
  | xs::xss -> xs          >>= (fun  y ->
                 expand xss >>= (fun ys ->
 return (y::ys)))
;;
let squash g = expand (List.map expand g);;
```

En utilisant cette fonction, il est facile de convertir une grille contenant des choix, en une liste de grilles sans choix:

```
let squash m = expand (List.map expand m);;
```

Et on obtient enfin un premier solveur (naïf) pour les Sudoku:

```
let solve k g = g |> choices k |> simplify k |> squash |> List.filter (valid k);;
```

```
solve 2 easy;;
- : int list list list =
[[[1; 3; 2; 4];
  [2; 4; 1; 3];
  [3; 1; 4; 2];
  [4; 2; 3; 1]];
 [[1; 3; 2; 4];
  [2; 4; 1; 3];
  [3; 2; 4; 1];
  [4; 1; 3; 2]]]
```

Exercice 7 (Extension (optionnel, points supplémentaires)) *Cette stratégie de résolution est très simpliste: après la première étape de simplification, on expanse tous les choixs, sans se soucier de leur compatibilité. Avec n valeurs possibles pour chacune des v cases vides, on peut se retrouver avec v^n matrices à tester ensuites.*
Pour reduire l'espace de recherche, il faudrait alterner expansion d'un choix avec simplification, en gardant les matrices qui n'ont plus de choix, et en jetant les matrices qui ne peuvent pas être complétées en une grille valide.
Supposons de disposer des fonctions suivantes: `blocked: int list matrix -> bool` *qui indique si une matrice ne peut être complétée;* `complete: int list matrix -> bool` *qui indique si une matrice ne contient plus de choix (toutes les entrées sont les listes contenant un seul valeur). Proposez une fonction plus efficace pour resoudre nos Sudokus.*

Solution 8 *Ici on vous donne aussi les fonctions qui n'etaient pas demandées pour l'exercice, afin que vous puissiez tester le code complet.*

```
(** solver with stepwise refinement *)

(* code for the function assumed in the exercise *)

let exists p l = List.fold_left (fun acc x -> (p x) || acc) false l;;

let complete g = forall (forall single) g;;

let void g = exists (exists (fun l -> l = [])) g;;

let consistent g = alldifferent (List.flatten (List.filter single g));;

let safe boxsize g =
  forall consistent (rows g) &&
  forall consistent (cols g) &&
  forall consistent (boxes boxsize g);;

let blocked blocksize g = void g || not (safe blocksize g);;
```

Pour faire l'expansion de façon incrémentale, il est nécéssaire de pouvoir identifier facilement la première case qui peut être expansée. Pour cela on utilise une fonction auxiliaire `break` *qui coupe une liste en deux au premier élémnent satisfaisant un prédicat* `p` *donné.*

```
(* code that was to be written to get the additional points *)

(* break a list in two at the first occurrence of an element satisfying p *)

let break p l =
  let rec aux acc =
    function
[] -> acc,[]
      | a::r as l when p a -> acc,l
      | a::r -> aux (a::acc) r
  in let l',l'' = aux [] l in List.rev l',l''
;;
```

Ensuite, une étape incrémentale d'expansion transforme une matrice contenant des choix en une liste de matrices dont le premier choix a été expansé.

```
let incexpand cm =
  let rows1,row::rows2 = break (exists (fun x -> not (single x))) cm in
  let row1,cs::row2 = break (fun x -> not (single x)) row in
  cs >>= (fun c ->
    return (rows1@
            [row1@([c]::row2)]@
            rows2))
;;
```

Et maintenant on peut enfin écrire la fonction recursive `search` *qui alterne expansion et simplification, ce qui nous permet d'obtenir notre solveur incrémental.*

```
let rec search boxsize m =
  if blocked boxsize m then []
  else if complete m then squash m
  else incexpand m >>= (fun m' ->
       search boxsize (simplify boxsize m'))
;;

let quicksolve boxsize g =
    g |> choices boxsize
      |> simplify boxsize
      |> search boxsize
;;
```