Master d'Informatique
Année 2011-2012
Examen de Programmation Fonctionnelle Avancée

Tous documents autorisés

Durée: 2h45

Première Session: 21 Mai 2012

Zippers (7 points)

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

On cherche à représenter des multiensembles, des ensembles dans lesquels chaque élément peut apparaître un certain nombre de fois.
Pour cela, nous utilisons la structure de donnée suivante, qui décrit des arbres binaires qui contiennent dans les noeuds des valeurs avec une multiplicité.

```
type 'a mset = Empty | Node of int * 'a * 'a mset * 'a mset;;
// type 'a mset = Empty | Node of int * 'a * 'a mset * 'a mset
```

Par exemple Node(3,"a",Empty,Empty) représente un multiensemble contenant 3 copies de "a".

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;*
3. *écrivez les fonctions permettant d'insérer et supprimer une valeur v, en traitant aussi le cas où le nœud courant contient déjà (un certain nombre de copies de) cette valeur.*

Typage (7 points)

Exercice 2 *Programmation sure : types fantômes*
Dans une librairie, vous trouvez le fragment de code suivant.

```
module Atom : sig
    type atom
    val mkInt : int -> atom
    val mkBool : bool -> atom
    val toString : atom -> string
    val double : atom -> atom
    val conjunction : atom -> atom -> atom
end =
struct
type atom = I of int | B of bool

let mkInt (i: int) = I i
let mkBool (b: bool) = B b

let toString = function
 | I i -> string_of_int i
 | B b -> string_of_bool b

let double (v: atom): atom = match v with
 | I i -> I (2* i)
 | _ -> failwith "type mismatch"

let conjunction (v1: atom) (v2: atom): atom =
match (v1,v2) with
 | B b1, B b2 -> B (b1 && b2)
 | _ -> failwith "type mismatch"
end;;
```

Vous vous apercevez qu'on peut facilement se tromper, et tomber sur des exceptions à l'exécution, par exemple en écrivant

```
open Atom
let x = mkBool true;;
// val x : Atom.atom = <abstr>

let y = double x;;
// Exception: Failure "type mismatch".
```

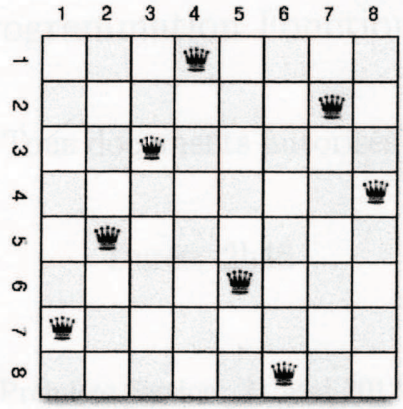
On vous demande d'utiliser le système de type pour éviter ces problèmes, et cela de deux façons:

1. *modifiez le code, en utilisant des types fantôme, de telle sorte que les exceptions ne soient jamais levées à l'exécution*
2. *modifiez le code, en utilisant des GADT, de telle sorte que l'on n'aie plus besoin de failwith dans le code*

Compréhension de listes (7 points)

Exercice 3 *Les N Reines*

Le problème des 8-reines est bien connu : il s'agit de trouver toutes les façons possibles de placer 8 reines sur un échiquier 8×8 de telle sorte qu'aucune reine soit en prise; sur la figure, vous voyez une de ces solutions.



Nous voulons programmer une extension de ce problème pour des échiquier de taille $n \times n$.

Pour cela, nous adoptons une représentation très compacte : une configuration de l'échiquier de taille $n \times n$ est donnée par une liste de n entiers [l1 ; l2 ;...; ln], avec l'entier lj donnant la ligne sur laquelle se trouve la reine de la colonne j. Donc, la configuration de la figure est [7;5;3;1;6;8;2;4] .
Avec cette représentation, il est facile de vérifier les conditions pour que les reines ne soient pas en prise :

- *on ne peut pas avoir deux reines dans la même colonne, par construction*
- *deux reines en li et lj sont sur la même ligne ssi li =lj*
- *deux reines en li et lj sont sur la même diagonale ssi abs(li−lj)= abs(i−j)*

On vous demande de :

1. *écrire une fonction compatible : int list −> int −> bool telle que l'appel compatible config l0 returne true si on peut ajouter une reine sur l'échiquier sur la ligne l0 dans la colonne immédiatement à gauche de celles de config, et false sinon*
2. *écrire une fonction nreines : int −> int list list qui prend un entier n en paramètre et calcule toutes les configurations légales avec n reines sur un échiquier de taille $n \times n$*
3. *commenter l'efficacité de la solution que vous avez proposée; est-il possible de réduire l'espace de recherche en modifiant un peu votre code[1] ?*

Vous pouvez utiliser la compréhension de listes dans votre solution, ou la monade des listes, à votre convenance, ainsi que le fonction auxiliaire range : int −> int list sans besoin de la redéfinir.

[1] Il se peut aussi qu'il soit déjà parfaitement efficace.