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

Tous documents autorisés

Durée: 2h45

Deuxième Session: 27 Juin 2013

Zippers (6 points)

Exercice 1 *Navigation de fichiers HTML à l'aide de zippers*
On utilise la structure de données simplifiée suivante pour représenter des fichiers HTML

```
type expr = P of string | Div of string * expr | Seq of expr * expr;;
```

Par exemple

```
<html>
  <div class="a">
    <p> Premier paragraphe </p>
    <div class="b">
      <p> Deuxieme paragraphe </p>
    </div>
  </div>
</html>
```

est representé par

```
Div ("a", Seq (P "Premier paragraphe",
               Div ("b",P "Deuxieme paragraphe"))
    );;
```

Afin d'utiliser des zippers pour naviguer dans ces structures

1. *dérivez formellement le type d'un zipper pour cette structure;*
2. *écrivez les fonctions permettant de se déplacer dans la structure: en haut, en bas (pour P et Div), en bas à droite et à gauche (pour Seq).*

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

```
  type bloc =  InDiv of string | InSeqR of expr | InSeqL of expr;;
// type bloc = InDiv of string | InSeqR of expr | InSeqL of expr


  type pile = bloc list;;
// type pile = bloc list


  type z_expr = pile * expr;;
// type z_expr = pile * expr


  exception NonDiv;;
// exception NonDiv

  exception NonSeq;;
// exception NonSeq

  exception EnBas;;
// exception EnBas

  exception EnHaut;;
// exception EnHaut


  let en_bas (pile, e) : z_expr = match e with
    | Div(s,x) -> (InDiv s)::pile, x
    | _ -> raise NonDiv;;
// val en_bas : bloc list * expr -> z_expr = <fun>


  let bas_gauche (pile, e) : z_expr = match e with
    | Seq(x, y) -> (InSeqL y)::pile, x
    | _ -> raise NonSeq;;
// val bas_gauche : bloc list * expr -> z_expr = <fun>


  let bas_droite (pile, e) : z_expr = match e with
    | Seq(x, y) -> (InSeqR x)::pile, y
    | _ -> raise NonSeq;;
// val bas_droite : bloc list * expr -> z_expr = <fun>


  let en_haut (pile, e) : z_expr = match pile with
    | (InDiv s)::p -> (p, Div(s, e))
    | (InSeqL e')::p -> (p, Seq (e,e'))
    | (InSeqR e')::p -> (p, Seq (e',e))
    | _ -> raise EnHaut;;
// val en_haut : bloc list * expr -> z_expr = <fun>
```

Visite d'une structure (7 points)

Exercice 2 (Manipulation de documents HTML) *Sur les documents définis comme dans l'exercice précédent, on vous demande d'écrire*

1. *une fonction replace : expr -> string -> string -> expr qui visite un document, en utilisant les zippers, et remplace les paragraphes contenant la première chaîne de caractères avec des paragraphes contenant la deuxième chaîne de caractères;*

2. *une fonction compare : expr -> expr -> bool qui prend en entrée deux documents, et retourne vrai si ils contiennent les mêmes paragraphes, dans les même ordre, et faux sinon (suggestion: n'utilisez pas des zippers pour cette partie, vous aurez plutot besoin d'une pile pour chaque document pour guider la visite)*

Solution 2 *Le remplacement se fait naturellemnt avec une visite recursive*

```
(* unfold zipper *)

let dezip out z =
  let rec aux = function
    | ([],e) -> e
    | (p,e) as z -> aux (out z)
  in aux z;;
// val dezip : ('a list * 'b -> 'a list * 'b) -> 'a list * 'b -> 'b = <fun>
//


(* replacement function *)

let replace a s s' =
  let rec aux z = match z with
    p,P v -> if v=s then (p,P s') else z
  | p,Div _ -> aux (en_bas z)
  | p,Seq(fg,fd) -> let z' = en_haut (aux (bas_gauche z)) in aux (bas_droite z')
  in dezip en_haut (aux ([],a))
;;
// val replace : expr -> string -> string -> expr = <fun>
```

Pour la comparaison, nous devons programmer une visite de deux structures. Pour guider la visite, nous nous servons d'une pile contenant les noeuds qui restent à visiter

```
let compare a a' =
  let rec pdiff = function
      (P v)::p,(P v')::p' -> if v = v' then pdiff (p,p') else false
    | a,(Seq(t,t'))::p' -> pdiff (a,(t::t'::p'))
    | a,(Div(_,t))::p' -> pdiff (a,(t::p'))
    | (Seq(t,t'))::p, a' -> pdiff ((t::t'::p),a')
    | (Div(_,t))::p, a' -> pdiff ((t::p),a')
    | [],[] -> true
    | _ -> false
  in pdiff ([a],[a'])
;;
// val compare : expr -> expr -> bool = <fun>
```

Il aurait aussi été possible de programmer la comparaison en construisant d'abord la liste des paragraphes, et en comparant les listes, mais cette approche est largement plus inefficace : si une

différence existe entre deux gros documents tout au début, on aimerait arreter la comparaison des qu'on la trouve, ce que fait la solution fournie ici.

Typage (7 points)

Exercice 3 *Programmation sure : types fantômes, GADT*

Dans une librairie, vous trouvez le fragment de code suivant.

```
module Expr : sig
    type e
    val mkInt : int -> e
    val mkString : string -> e
    val toString : e -> string
    val double : e -> e
    val concat : e -> e -> e
end =
struct
type e = I of int | S of string
let mkInt (i: int) = I i
let mkString (s: string) = S s
let toString = function
 | I i -> string_of_int i
 | S s -> s
let double (v: e): e = match v with
 | I i -> I (2* i)
 | _ -> failwith "type mismatch"
let concat (s1: e) (s2: e): e =
match (s1,s2) with
 | S s, S s' -> S (s^s')
 | _ -> failwith "type mismatch"
end;;
```

On peut facilement se tromper, et tomber sur des exceptions à l'exécution, par exemple en écrivant

```
open Expr
let x = mkString "3";;
// val x : Expr.e = <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*

Solution 3 *Avec les types fantôme on écrirait:*

```
module Expr : sig
    type 'a e
    val mkInt : int -> int e
    val mkString : string -> string e
    val toString : 'a e -> string
```

```
    val double : int e -> int e
    val concat : string e -> string e -> string e
end =
struct
type 'a e = I of int | S of string
let mkInt (i: int) : int e= I i
let mkString (s: string) : string e = S s

let toString = function
 | I i -> string_of_int i
 | S s -> s

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

let concat (s1: string e) (s2: string e): string e =
match (s1,s2) with
 | S s, S s' -> S (s^s')
 | _ -> failwith "type mismatch"
end;;
// module Expr :
//   sig
//     type 'a e
//     val mkInt : int -> int e
//     val mkString : string -> string e
//     val toString : 'a e -> string
//     val double : int e -> int e
//     val concat : string e -> string e -> string e
//   end

open Expr;;

let x = mkString "3";;
// val x : string Expr.e = <abstr>

let y = double x;;
// Characters 15-16:
//   let y = double x;;
//                 ^
// Error: This expression has type string Expr.e
//        but an expression was expected of type int Expr.e
```

Notez qu'il aurait suffit de changer l'interface: j'ai rajouté les types aux fonctions dans le corps du module pour lisibilite, mais ce n'est pas nécessaire!

Avec les GADT:

```
type 'a e =
  I : int -> int e
| S : string -> string e;;
// type 'a e = I : int -> int e | S : string -> string e
```

```
let mkInt (i: int) = I i
let mkString (s: string) = S s;;
// val mkInt : int -> int e = <fun>
// val mkString : string -> string e = <fun>


let toString : type a. a e -> string = function
  | I i -> string_of_int i
  | S s -> s;;
// val toString : 'a e -> string = <fun>


let double (v: int e): int e = match v with
  | I i -> I (2*i);;
// val double : int e -> int e = <fun>


let concat (v1: string e) (v2: string e): string e =
match (v1,v2) with
  | S s, S s' -> S (s^s');;
// val concat : string e -> string e -> string e = <fun>


let x = mkString "3";;
// val x : string e = S "3"

let y = double x;;
// Characters 15-16:
//   let y = double x;;
//                 ^
// Error: This expression has type string e
//        but an expression was expected of type int e
```

Ici on doit dans tous les cas donner le type pour toString.