Examen. Session 1. Documents autorisés. Durée 1h30.

1 Typage avancé : arbres binaires hétérogènes

Sur le modèle des listes hétérogènes, on souhaite définir les arbres binaires hétérogènes. On rappelle ci-dessous quelques définitions :

```
(* type des listes et arbres binaires standard *)
type 'a list = | Nil : 'a list | Cons : 'a * 'a list -> 'a list
type 'a tree = | Vide : 'a tree | Noeud : 'a tree * 'a * 'a tree -> 'a tree
(* type des listes hétérogènes *)
type _ hlist = | HNil : unit hlist | HCons : 't * 'q hlist -> ('t * 'q) hlist
```

Exercice 1 (Type) Définir le type _ htree des arbres binaires hétérogènes. On doit avoir par exemple : HNoeud (HVide, 1, HNoeud (HVide, 2, HVide)) : (unit * int * (unit * int * unit)) htree

Solution 1

```
type _ htree =
  | HVide : unit htree
  | HNoeud : 'g htree * 'r * 'd htree -> ('g * 'r * 'd) htree
```

Exercice 2 (Opérations) Donner les types et définir les fonctions suivantes : left, right, root, qui respectivement renvoient le sous-arbre gauche, le sous-arbre droit et la racine d'un arbre binaire hétérogène non vide (garanti par typage).

Solution 2

```
(* on a besoin d'un schéma de type explicite pour garantir que l'arbre est non vide *)
(* les fonctions sont non récursives, donc pas besoin d'un schéma non uniforme     *)
let left : ('g * _ * _) htree -> 'g htree = function
  | HNoeud (g, _, _) -> g

let right : (_ * _ * 'd) htree -> 'd htree = function
  | HNoeud (_, _, d) -> d

let root : (_ * 'r * _) tree -> 'r = function
  | HNoeud (_, r, _) -> r
```

Le type _ htree ne se prête pas simplement à l'écriture de fonctions récursives. On souhaite définir le type des arbres binaires hétérogènes parfaits, i.e. tels qu'à chaque noeud, les sous-arbres gauche et droit aient le même profil, i.e. soient superposables.

Exercice 3 (Type) Définir le type _ hperfect des arbres binaires hétérogènes parfaits.

Solution 3

```
type _ hperfect =
  | HVide : unit hperfect
  | HNoeud : 'p htree * 'r * 'p htree -> ('r * 'p) htree (* ou ('p * 'r * 'p) hperfect *)
```

Exercice 4 (Opérations) Donner les types et définir les fonctions suivantes :

1. mirror, qui inverse les branches gauches et droites de tous les noeuds d'un arbre _ hperfect.
2. depth, qui calcule efficacement la profondeur d'un arbre _ hperfect.

Solution 4

```
let rec mirror : type p. p hperfect  -> p hperfect = function
  | HVide              -> HVide
  | HNoeud (g, r, d) -> HNoeud (mirror d, r, mirror g)

let rec depth : type p. p hperfect -> int = function
  | HVide              -> 0
  | HNoeud (l, _, _) -> depth l
```

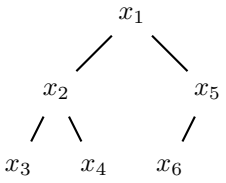
2 Curseur d'arbres binaires

La structure de curseur d'arbres permet de se déplacer dans un arbre et de modifier l'élément courant sans devoir reconstruire l'arbre entier. On se donne les types suivants :

```
(* type des arbres binaires standard *)
type 'a tree = Noeud of ('a tree * 'a * 'a tree) | Vide
(* type des chemins dans un arbre = branche gauche/droite, élément rencontré, branche non prise *)
type 'a path = (bool * 'a * 'a tree) list
(* type des curseurs = sous-arbre courant, chemin "à l'envers" depuis son père jusqu'à la racine *)
type 'a curs = 'a tree * 'a path
```

Par exemple, un curseur à la position x_4 dans l'arbre ci-dessous (où les branches Vide sont omises), sera représenté par la paire :

```
  (Noeud(Vide, x4, Vide),
  [ (true, x2, Noeud(Vide, x3, Vide)); (false, x1, Noeud(Noeud(Vide, x6, Vide), x5, Vide)) ])
```



Exercice 5 (Opérations locales) Définir les fonctions suivantes :

1. init : 'a tree -> 'a curs qui crée un curseur positionné à la racine de l'arbre.
2. get : 'a curs -> 'a qui renvoie l'élément à la position courante du curseur (si non Vide).
3. set : 'a curs -> 'a -> 'a curs qui remplace l'élément à la position courante du curseur (si non Vide) par celui passé en paramètre.

Solution 5

```
let init t = t, []

let get = function
  | (Noeud (_, r, _), _) -> r
  | _                    -> failwith "arbre vide"

let set = function
  | Noeud (g, _, d), p) -> (fun v -> (Noeud (g, v, d), p))
  | _                   -> failwith "arbre vide"
```

Exercice 6 (Navigation) Définir les fonctions suivantes :

1. left / right : 'a curs -> 'a curs qui permettent, si les branches existent, d'aller au fils gauche / droit respectivement, à partir d'une position donnée du curseur.

2. back : 'a curs -> 'a curs qui permet de remonter au noeud père de la position courante (sauf si on est à la racine).

Solution 6

```
let left = function
  | (Noeud (g, r, d), p) -> (g, (true , r, d)::p)
  |                      -> failwith "arbre vide"

let right = function
  | (Noeud (g, r, d), p) -> (d, (false, r, g)::p)
  |                      -> failwith "arbre vide"

let back (t, p) =
  match p with
  | []                  -> failwith "pas de père"
  | (true , r, d)::q -> (Noeud (t, r, d), q)
  | (false, r, g)::q -> (Noeud (g, r, t), q)
```

On s'intéresse à des fonctions de navigation plus complexes. On veut pouvoir marquer la position courante dans l'arbre, pour pouvoir y revenir en une seule opération, après avoir exploré les sous-arbres. Ces fonctionnalités supplémentaires ne doivent pas affecter le comportement des opérations left , right et back.

Exercice 7 (Nouveau type des chemins) Redéfinir le type 'a path, sachant qu'un élément de ce chemin peut maintenant être soit un marqueur, soit un triplet (bool * 'a * 'a tree) comme précédemment.

Solution 7

```
type 'a elt = | Mark | Triple of bool * 'a * 'a tree
type 'a path = 'a elt list
type 'a curs = 'a tree * 'a path
```

Exercice 8 (Navigation avancée) Avec le nouveau type 'a path, (re)définir les fonctions suivantes :

1. left / right / back : 'a curs -> 'a curs
2. mark : 'a curs -> 'a curs qui permet d'insérer dans le chemin courant le marqueur.
3. exit : 'a curs -> 'a curs qui permet de remonter tous les noeuds ancêtres de la position courante, jusqu'à ce qu'on rencontre un marqueur (ou la racine). Le marqueur est alors retiré.

Solution 8

```
  let left = function
  | (Noeud (g, r, d), p) -> (g, (Triple (true , r, d))::p)
  | _                    -> failwith "arbre vide"

let right = function
  | (Noeud (g, r, d), p) -> (d, (Triple (false, r, g))::p)
  | _                    -> failwith "arbre vide"

let rec back (t, p) =
  match p with
  | []                              -> failwith "pas de père"
  | (Triple (true , r, d))::q -> (Noeud (t, r, d), q)
  | (Triple (false, r, g))::q -> (Noeud (g, r, t), q)
  | Mark                      ::q -> back (t, q)
```