Université Paris Diderot
Informatique

Programmation Fonctionnelle
Année 2016-2017

TP n°3 bis - Correction

Arbres binaires – Bonus

1 Itérateur d'arbres

Exercice 1. Sans déclarer de fonctions auxiliaires et sans vous servir des fonctions prédéfinies, écrire :

`forall_labels : ('a -> bool) -> 'a tree -> bool`

telle que `forall_labels p a` renvoie `true` si et seulement si chaque étiquette `x` de `a` vérifie `(p x) = true`. A partir de cette seule fonction, écrire une fonction

`is_uniform : 'a -> 'a tree -> bool`

tel que `is_uniform v a` renvoit `true` si et seulement si chaque étiquette de `a` est égale à `v`. Cette fonction ne devra pas être déclarée comme récursive, mais doit déléguer entièrement la récurrence à la précédente.
Correction :

```
let rec forall_labels p a = match a with
    Nil -> true
  | Node(x, g, d) -> p x && (forall_labels p g) && (forall_labels p d)
;;

let is_uniform x a = forall_labels (fun n -> n = x) a;;
```

Exercice 2. Toujours sans fonctions auxiliaires ou prédéfinies, écrire :

`forall_subtrees : ('a -> 'a tree -> 'a tree -> bool) -> 'a tree -> bool`

telle que `forall_subtrees p a` renvoie `true` si et seulement si pour chaque sous-arbre de `a` de la forme `Node(x, g, d)`, on a `p x g d = true`.

Rappelons qu'un arbre est un *peigne droit* s'il est réduit à l'arbre vide, ou si son fils gauche est vide et son fils droit est un peigne droit. A partir de la seule fonction `forall_subtrees`, écrire :

`est_peigne_droit : 'a tree -> bool`

tel que `est_peigne_droit a` renvoit `true` si et seulement si `a` est un peigne droit, et déléguant entièrement la récurrence à la fonction `forall_subtrees`.
Correction :

```
let rec forall_subtrees pn a = match a with
    Nil -> true
  | Node(x, g, d) ->
      pn x g d &&
     (forall_subtrees pn g) &&
```

```
      (forall_subtrees pn d)
;;

let est_peigne_droit a =
  forall_subtrees (fun _ g _ -> g = Nil) a;;
;;
```

Exercice 3. On définit pour les arbres un itérateur de la manière suivante :

```
let rec fold_tree fn vf a = match a with
    Nil -> vf
  | Node(n, g, d) -> fn n (fold_tree fn vf g) (fold_tree fn vf d);;
```

Noter que `vf` spécifie la valeur à renvoyer si l'arbre `a` est réduit à un arbre vide. Si c'est un noeud interne, l'itérateur est appliqué aux sous-arbres avec les mêmes arguments, et les valeurs de retour sont combinées à la valeur étiquetant la racine via la fonction `fn`.

Quel est le type de `fold_tree` ?

Exercice 4. En utilisant uniquement `fold_tree`, et en lui déléguant la récurrence, écrire :

```
somme_etiquettes : int tree -> int
```

renvoyant la somme des étiquettes d'un arbre étiqueté par des entiers.

Avec les mêmes contraintes, écrire :

```
map_tree : ('a -> 'b) -> 'a tree -> 'b tree
```

telle que `map_tree f a` renvoie l'arbre obtenu en remplaçant chaque étiquette `x` de `a` par `(f x)`.

Correction :

```
let somme_labels a =
  fold_up (fun n sg sd -> n + sg + sd) 0 a
;;
let map_tree f a =
  fold_up (fun n gm gd -> Node(f n, gm, gd)) a
;;
```

2 Arbres binaires de recherche – exercices avancés

Exercice 5. Écrire la fonction suivante :

```
retire_abr : int -> int_tree -> int_tree
```

tel que, si `x` est un entier et `a` est un arbre, `retire_abr x a` est un arbre binaire de recherche qui ne contient pas `x`. Si `a` ne contient pas `x`, on retournera l'arbre inchangé.

Attention à préserver la propriété des ABRs. Vous testerez votre fonction sur l'exemple suivant :

```
retire_abr 3 (Node 3 (Node 1 Nil (Node 2 Nil Nil)) (Node 4 Nil Nil))
```

Correction :

```
let rec retire_min a =
   match a with
     Nil -> failwith "arbre vide"
```

```
   | Node(v,Nil,d) -> v,d
   | Node(v,g,d) -> match retire_min g with
                     x,a -> x,Node(v,a,d);;
let rec retire_abr x a =
   match a with
     Nil -> Nil
   | Node(v,g,d) when x < v -> Node (v,(retire_abr x g),d)
   | Node(v,g,d) when x > v -> Node (v,g,(retire_abr x d))
   | Node(v,Nil,Nil) -> Nil
   | Node(v,g,Nil) -> g
   | Node(v,Nil,d) -> d
   | Node(v,g,d) -> match (retire_min d) with
                     x,d' -> Node(x,g,d');;
```

Exercice 6. On défini le facteur de balancement d'un ABR a de la façon suivante :

$$balancement(a) = hauteur(sous\_arbre\_droit(a)) - hauteur(sous\_arbre\_gauche(a))$$

Un ABR est dit balancé si $|balancement(a)| <= 1$.

1. En quoi un ABR balancé est il préférable a un ABR non balancé ? On considérera le coût de la fonction `contient_abr`.

(⋆⋆) 2. Écrire la fonction `balance_abr : int_tree -> int_tree` tel que, étant donné un ABR non balancé `a`, `balance_abr a` est un ABR balancé.