Haskell (IR3) – Arbres Binaires

Stéphane Vialette

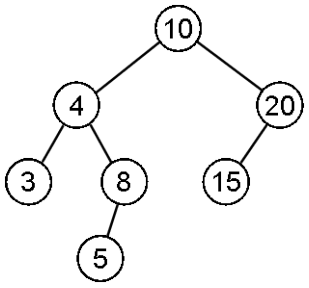
20 octobre 2020

Question 1: Arbres binaires

Considérons la variante suivante du type récursif des arbres binaires sur les entiers, défini par :

```
data BTree a = Empty | Branch (BTree a) a (BTree a)
               deriving (Show)
```

(a) Écrire une expression Haskell, de type `BTree`, qui représente l'arbre suivant.



Pour simplifier, dans la suite, nous supposerons que cet arbre est retourné par la fonction `exampleBT :: BTree Int`.

Attention, si vous laissez haskell inférérer le typage de la fonction, vous aller sans doute obtenir `exampleBT :: BTree Integer`. Mais le type `Integer` n'implémente pas la classe `Bounded` (c'est bien normal pour des entiers arbitrairement longs) alors que cette contrainte de classe est nécéssaire dans la suite du TP.

Les fonctions haskell suivantes vous permettrons de visualiser plus facilement vos arbres :

```
indent :: Int -> String
indent = flip L.replicate '.'

str :: (Show a) => BTree a -> String
str = aux 0
  where
    aux k Empty             = indent k ++ "⊥\n"
    aux k (Branch lt x rt) = indent k ++ show x ++ "\n" ++
                             aux (k+1) lt ++ aux (k+1) rt
```

Nous pouvons les utiliser de la façon suivante :

```
ghci> let s = str exampleBT in putStrLn s
10
.4
..3
...⊥
...⊥
..8
...5
....⊥
....⊥
...⊥
.20
..15
...⊥
...⊥
..⊥
```

Solution:
On peut définir l'arbre pas à pas dans `ghci` (pas très pratique !).

```
ghci> bt3  = Branch Empty  3 Empty
ghci> bt5  = Branch Empty  5 Empty
ghci> bt8  = Branch bt5    8 Empty
ghci> bt4  = Branch bt4    4 bt8
ghci> bt15 = Branch Empty 15 Empty
ghci> bt20 = Branch bt15  20 Empty
ghci> bt10 = Branch bt4   10 bt20
```

Le mieux est de définir cet example via une fonction pour tester rapidement les fonctions que vous allez écrire.

```
exampleBT :: BTree Int
exampleBT = bt10
  where
    bt3  = Branch Empty 3 Empty
    bt5  = Branch Empty 5 Empty
    bt8  = Branch bt5   8 Empty
    bt4  = Branch bt3   4 bt8
    bt15 = Branch Empty 15 Empty
    bt20 = Branch bt15  20 Empty
    bt10 = Branch bt4   10 bt20
```

(b) Écrire la fonction

```
emptyBT :: BTree a
```

qui retourne un arbre binaire vide.

Solution:
La fonction `emptyBT :: BTree a` n'a pas pour but de remplacer l'utilisation du constructeur `Empty` du type `BTree a`. Voyez la fonction `emptyBT :: BTree a` comme une fonction retournant un arbre binaire particulier.
La fonction ne présente de toute façon aucune difficulté.

```
emptyBT :: BTree a
```

```haskell
emptyBT = Empty
```

(c) Écrire la fonction

```haskell
size :: Num b => BTree a -> b
```

qui retourne le nombre de sommets dans un arbre binaire.

Solution:

Une solution récursive immédiate.

```haskell
size :: Num b => BTree a -> b
size Empty          = 0
size (Branch l _ r) = 1 + size l + size r
```

Une autre solution récursive qui utilise un accumulateur.

```haskell
size' :: Num b => BTree a -> b
size' = aux 0
  where
    aux acc Empty          = acc
    aux acc (Branch l _ r) = aux (1 + acc') r
      where
        acc' = aux acc l
```

(d) Écrire la fonction

```haskell
maxBT :: (Ord a, Bounded a) => BTree a -> a
```

qui retourne l'étiquette maximale dans un arbre binaire. Dans un premier temps (et pour simplifier), nous supposerons que la fonction retourne `minBound :: a` pour l'arbre vide.

Solution:

```haskell
maxBT' :: (Ord a, Bounded a) => BTree a -> a
maxBT' Empty = minBound
maxBT' (Branch l x r)
  | x    >= max lMax rMax = x
  | lMax >= max x    lMax = lMax
  | otherwise             = rMax
  where
    lMax = maxBT l
    rMax = maxBT r
```

On préférera une solution plus compacte (et plus lisible il me semble).

```haskell
maxBT :: (Ord a, Bounded a) => BTree a -> a
maxBT Empty = minBound
maxBT (Branch l x r) = maximum [maxBT l, x, maxBT r]
```

Attention à ne pas confondre les fonctions `max :: Ord a => a -> a -> a` et `maximum :: (Foldable t, Ord a) => t a -> a`.

(e) Écrire la fonction

```haskell
minBT :: (Ord a, Bounded a) => BTree a -> a
```

qui retourne l'étiquette minimale dans un arbre binaire. Dans un premier temps (et pour simplifier), nous supposerons que la fonction retourne `maxBound :: a` pour l'arbre vide.

Solution:
Pas de difficultés ici, on reprend l'idée utilisée pour la fonction `maxBT`.

```
maxBT :: (Ord a, Bounded a) => BTree a -> a
maxBT Empty          = maxBound
maxBT (Branch l x r) = maximum [maxBT l, x, maxBT r]
```

(f) Pouvez vous extraire la logique des fonctions `maxBT` et `minBT` et factoriser le code commun ?

Solution:
Effectivement, les fonctions `maxBT` et `minBT` semblent partager la même logique. Plus précisement, la différence réside dans l'utilisation des fonctions `maximum` et `minimum`, et des constantes `minBound` et `maxBound` .

```
app :: ([a] -> a) -> a -> BTree a -> a
app _ z Empty          = z
app f z (Branch l x r) = f [app f z l, x, app f z r]

minBT' :: (Ord a, Bounded a) => BTree a -> a
minBT' = app minimum maxBound

maxBT' :: (Ord a, Bounded a) => BTree a -> a
maxBT' = app maximum minBound
```

(g) Clairement, l'utilisation des fonctions `maxBound` et `minBT''` n'est pas satisfaisante. Écrire maintenant les fonctions

```
maxBT'' :: BTree a -> Maybe a
minBT'' :: BTree a -> Maybe a
```

qui retournent les étiquettes maximales et minimales (respectivement) d'un arbre binaire. Cette fois ci, la valeur maximale et la valeur minimale de l'arbre vide est **`Nothing`**.

```
ghci> maxBT'' emptyBT
Nothing
ghci> maxBT'' exampleBT
Just 20
ghci> minBT'' emptyBT
Nothing
ghci> minBT'' exampleBT
Just 3
ghci>
```

Solution:
On remarque que :

```
ghci> maximum [Nothing]
Nothing
ghci> maximum [Nothing, Just 1]
```

```
Just 1
ghci> maximum [Just 2, Nothing, Just 1]
Just 2
ghci> maximum [Just 2, Nothing, Just 1, Nothing, Just 3]
Just 3
```

et

```
ghci> minimum [Nothing]
Nothing
ghci> minimum [Nothing, Just 1]
Nothing
ghci> minimum [Just 2, Nothing, Just 1]
Nothing
ghci>  minimum [Just 2, Just 3, Just 1]
Just 1
```

Donc,

— `Nothing < Just x` quelque soit `x`.
— `Just x < Just y` si et seulement si `x < y`.

On peut maintenant en déduire facilement les fonctions `maxBT''` et `minBT''` (attention à l'écriture de `minBT''`, dès que l'on retourne `Nothing` on a un minimum)

```
import qualified Data.Maybe as Maybe

maxBT'' :: (Ord a) => BTree a -> Maybe a
maxBT'' Empty          = Nothing
maxBT'' (Branch l x r) = maximum [maxBT'' l, Just x, maxBT'' r]

minBT'' :: (Ord a) => BTree a -> Maybe a
minBT'' Empty                  = Nothing
minBT'' (Branch Empty x Empty) = Just x
minBT'' (Branch l     x Empty) = minimum [minBT'' l, Just x]
minBT'' (Branch Empty x r)     = minimum [           Just x, minBT'' r]
minBT'' (Branch l     x r)     = minimum [minBT'' l, Just x, minBT'' r]
```

Une version moins élégante de `minBT''` mais a le mérite de mettre en lumière la petite dificulté.

```
import qualified Data.Maybe as Maybe

minBT'' :: (Ord a) => BTree a -> Maybe a
minBT'' Empty          = Nothing
minBT'' (Branch l x r) = minimum xs
  where
    xs = filter keepJust [minBT'' l, Just x, minBT'' r]
      where
        keepJust Nothing = False
        keepJust _       = True
```

(h) Écrire la fonction

```
height :: (Ord b, Num b) => BTree a -> b
```

qui retourne la hauteur de l'abre binaire. La hauteur de l'arbre vide est 0.

Solution:

Aucune difficulté ici.

```
height :: (Ord b, Num b) => BTree a -> b
height Empty          = 0
height (Branch l _ r) = 1 + max (height l) (height r)
```

(i) Écrire la fonction

```
searchBT :: Eq a => BTree a -> a -> Bool
```

qui retourne `True` si un entier donné apparaît dans un abre binaire.

Solution:

Nos arbres binaires ne sont pas des arbres binaires de recherche (pas encore !). En conséquence, une valeur peut se trouver à la racine, dans le sous-arbre gauche ou dans le sous-arbre droit.

```
searchBT :: Eq a => BTree a -> a -> Bool
searchBT Empty _ = False
searchBT (Branch l x r) y
  | x == y       = True
  | otherwise    = searchBT l y || searchBT r y
```

On pourra même préférer la version suivante qui met en avant une logique basée explicitement sur les disjonctions.

```
searchBT' :: Eq a => BTree a -> a -> Bool
searchBT' Empty _          = False
searchBT' (Branch l x r) y = x == y || searchBT' l y || searchBT' r y
```

(j) Écrire la fonction

```
toList :: BTree a -> [a]
```

qui retourne la liste des éléments qui apparaissent dans un abre binaire. L'ordre des éléments n'est pas contraint. Par exemple,

```
ghci> toList exampleBT
[10,4,3,8,5,20,15]
```

Solution:

```
toList :: BTree a -> [a]
toList Empty          = []
toList (Branch l x r) = x : toList l ++ toList r
```

(k) Écrire les fonctions

```
preVisit :: BTree a -> [a]
inVisit :: BTree a -> [a]
postVisit :: BTree a -> [a]
```

qui retournent les éléments d'un arbre binaire par un parcours préfixe, infixe et suffixe. On pourra en profiter pour réécrire la fonction `toList`.

```
ghci> preVisit exampleBT
[10,4,3,8,5,20,15]
ghci> inVisit exampleBT
[3,4,5,8,10,15,20]
ghci> postVisit exampleBT
[3,5,8,4,15,20,10]
```

Solution:

```haskell
preVisit :: BTree a -> [a]
preVisit Empty          = []
preVisit (Branch l x r) = x : preVisit l ++ preVisit r

inVisit :: BTree a -> [a]
inVisit Empty          = []
inVisit (Branch l x r) = inVisit l ++ [x] ++ inVisit r

postVisit :: BTree a -> [a]
postVisit Empty          = []
postVisit (Branch l x r) = postVisit l ++ postVisit r ++ [x]

toList :: BTree a -> [a]
toList = preVisit
```

(l) Écrire la fonction

```haskell
filterBT :: (a -> Bool) -> BTree a -> [a]
```

qui retourne la liste des éléments d'un arbre binaire filtrés par un prédicat. Encore une fois, l'ordre des éléments n'est pas contraint.

```
ghci> filterBT even exampleBT
[10,4,8,20]
ghci> filterBT odd exampleBT
[3,5,15]
ghci> filterBT (> 5) exampleBT
[10,8,20,15]
ghci> filterBT (< 0) exampleBT
[]
```

Solution:

On ne réinvente pas la poudre (on a maintenant `toList` et on connaît (très bien !) la fonction `filter :: (a -> Bool) -> [a] -> [a]`).

```haskell
filterBT :: (a -> Bool) -> BTree a -> [a]
filterBT f = filter f . toList
```

Plus généralement, cette fonction n'aurait pas vraiment de sens dans une API bien réfléchie. En effet, la fonction `toList :: BTree a -> [a]` est suffisante.

(m) Écrire la fonction

```haskell
mapBT :: (a -> b) -> BTree a -> BTree b
```

qui retourne un **nouvel** arbre binaire obtenu en appliquant une fonction sur chaque élément d'un arbre binaire. Par exemple,

```
ghci>  putStr . str $ mapBT (*100) exampleBT
1000
.400
..300
...⊥
...⊥
..800
...500
....⊥
....⊥
...⊥
.2000
..1500
...⊥
...⊥
..⊥
```

Solution:
Les données sont non mutables. Il faut donc reconstruire un arbre binaire.

```
mapBT :: (a -> b) -> BTree a -> BTree b
mapBT _ Empty = Empty
mapBT f (Branch l x r) = Branch l' x' r'
  where
    x' = f x
    l' = mapBT f l
    r' = mapBT f r
```

ou encore (pas forcement plus lisible ! ?)

```
mapBT' :: (a -> b) -> BTree a -> BTree b
mapBT' _ Empty          = Empty
mapBT' f (Branch l x r) = Branch (mapBT f l) (f x) (mapBT f r)
```

Question 2: Arbres binaires de recherche

Nous nous intéressons maintenant aux arbres binaires de recherche : les éléments dans le sous-arbre gauche sont inférieurs ou égaux à la racine, et ceux du sous-arbre droit sont strictement supérieurs.

(a) Écrire la fonction

```
insertBST :: (Ord a) => BTree a -> a -> BTree a
```

qui insert un élément dans un arbre binaire de recherche (les doublons sont autorisés).

Solution:

```
insertBST :: (Ord a) => BTree a -> a -> BTree a
insertBST Empty y = Branch Empty y Empty
insertBST (Branch l x r) y
  | y <= x    = Branch (insertBST l y) x r
  | otherwise = Branch l               x (insertBST r y)
```

Il est très important (fondamental !) d'observer que l'on reconstruit un arbre binaire à chaque appel.

(b) Écrire la fonction

```
searchBST :: (Ord a) => BTree a -> a -> Bool
```

qui retourne `True` si et seulement si un élément donné apparaît dans un abre binaire de recherche.

Solution:

```
searchBST :: (Ord a) => BTree a -> a -> Bool
searchBST Empty _           = False
searchBST (Branch l x r) y
  | x == y    = True
  | y < x     = searchBST l y
  | otherwise = searchBST r y
```

(c) Écrire la fonction

```
deleteLargestBST :: BTree a -> Maybe (a, BTree a)
```

qui retourne une paire contenant l'élément maximum dans un arbre binaire de recherche et l'arbre binaire de recherche obtenu en supprimant cet élément.

```
ghci> deleteLargestBST emptyBT
Nothing
ghci> res = deleteLargestBST exampleBT
ghci> import Data.Maybe
ghci> fst $ fromJust res
20
ghci> putStr . str . snd $ fromJust res
10
.4
..3
...⊥
...⊥
..8
...5
....⊥
....⊥
...⊥
.15
..⊥
..⊥
```

Solution:

```
deleteLargestBST :: BTree a -> Maybe (a, BTree a)
deleteLargestBST Empty                = Nothing
deleteLargestBST (Branch l x Empty) = Just (x, l)
deleteLargestBST (Branch l x r)       = case deleteLargestBST r of
 Nothing       -> Nothing
 Just (y, r') -> Just (y, Branch l x r')
```

(d) Écrire la fonction

```
deleteBST :: (Ord a) => BTree a -> a -> BTree a
```

qui supprime - s'il existe - un élément dans un arbre binaire de recherche. La fonction retourne l'arbre non modifié si l'élément recherché ne se trouve pas dans l'arbre binaire de recherche. Pensez à utiliser `deleteLargestBST`.

```
ghci> putStr . str $ deleteBST emptyBT 1
⊥
ghci> putStr . str $ deleteBST exampleBT 10
8
.4
..3
...⊥
...⊥
..5
...⊥
...⊥
.20
..15
...⊥
...⊥
..⊥
ghci>  putStr . str $ deleteBST exampleBT 4
10
.3
..⊥
..8
...5
....⊥
....⊥
...⊥
.20
..15
...⊥
...⊥
..⊥
```

Solution:

L'algorithme est le suivant : si on a trouvé l'élément que l'on doit supprimer, on peut aller chercher l'élément maximum du sous-arbre gauche pour le mettre à la place. La seule petit dfficulté est si le sous-arbre gauche est vide.

```haskell
deleteBST :: (Ord a) => BTree a -> a -> BTree a
deleteBST Empty _ = Empty
deleteBST bst@(Branch Empty x r) y
  | x == y    = r
  | y > x     = Branch Empty x (deleteBST r y)
  | otherwise = bst
deleteBST (Branch l x r) y
  | x == y    = Branch l' x' r
  | y < x     = Branch (deleteBST l y) x r
  | otherwise = Branch l x (deleteBST r y)
  where
    Just (x', l') = deleteLargestBST l
```

(e) Écrire la fonction

```
deleteBST' :: (Ord a) => BTree a -> a -> Maybe (BTree a)
```

qui supprime - s'il existe - un élément dans un arbre binaire de recherche. La fonction retourne `Nothing` si l'élément recherché ne se trouve pas dans l'arbre binaire de recherche. Encore une fois, pensez à utiliser `deleteLargestBST`.

Solution:
Une première solution de filou ... mais pas très efficace !

```
instance (Eq a) => Eq (BTree a) where
  Empty == Empty = True
  Empty == _     = False
  _     == Empty = False
  Branch l x r == Branch l' x' r' = x == x' && l == l' && r == r'

deleteBST' :: (Ord a) => BTree a -> a -> Maybe (BTree a)
deleteBST' bst y = if bst' /= bst then Just bst' else Nothing
  where
    bst' = deleteBST bst y
```

Une solution préférable est la suivante.

```
deleteBST :: (Ord a) => BTree a -> a -> Maybe (BTree a)
deleteBST Empty _ = Just Empty
deleteBST (Branch Empty x r) y
  | x == y    = Just r
  | y > x     = case deleteBST r y of
    Nothing -> Nothing
    Just r' -> Just (Branch Empty x r')
  | otherwise = Nothing
deleteBST (Branch l x r) y
  | x == y    = Just (Branch l' x' r)
  | y < x     = case deleteBST l y of
    Nothing -> Nothing
    Just l' -> Just (Branch l' x r)
  | otherwise = case deleteBST r y of
    Nothing -> Nothing
    Just r' -> Just (Branch l x r')
  where
    Just (x', l') = deleteLargestBST l
```

Question 3: (*) Pour aller (un peu) plus loin

(a) Ecrire la fonction

```
mkBST :: [a] -> BTree a
```

qui construit un arbre binaire de recherche à partir d'une liste d'éléments.

Solution:
On reconnaît tout de suite (?!) un fold.

```
import qualified Data.Foldable as F
```

```haskell
mkBST :: (Ord a) => [a] -> BTree a
mkBST = F.foldl insertBST emptyBT
```

ou encore

```haskell
mkBST :: (Ord a) => [a] -> BTree a
mkBST = F.foldr (flip insertBST) emptyBT
```

(b) Ecrire la fonction

```haskell
levelVisit :: BTree a -> [a]
```

qui retourne la liste des éléments d'un arbre binaire par niveau.

```
ghci> levelVisit exampleBT
[10,4,20,3,8,15,5]
```

Solution:

Pensons fonctionnelle. Il suffit d'indexer les éléments par leur profondeur et de trier ensuite par profondeur.

```haskell
import qualified Data.List   as L
import qualified Data.Tuple  as T

levelVisit :: BTree a -> [a]
levelVisit = render . indexByDepth 0 []
  where
    indexByDepth _ acc Empty          = acc
    indexByDepth i acc (Branch l x r) = indexByDepth (i+1) acc'' l
      where
        acc'  = (i, x) : acc
        acc'' = indexByDepth (i+1) acc' r
    render = L.map T.snd . L.sortBy cmp
      where
        (i, _) `cmp` (j, _) = i `compare` j
```

Une autre solution.

```haskell
import qualified Data.Foldable as F

zipWithPad :: [[a]] -> [[a]] -> [[a]]
zipWithPad []       []       = []
zipWithPad (x : xs) []       = x        : zipWithPad xs []
zipWithPad []       (y : ys) = y        : zipWithPad [] ys
zipWithPad (x : xs) (y : ys) = (x ++ y) : zipWithPad xs ys

levelVisit' :: BTree a -> [a]
levelVisit' = F.foldl (++) [] . aux
   where
     aux Empty          = []
     aux (Branch l x r) = [x] : zipWithPad (aux l) (aux r)
```