Haskell (IR3) – Arbres Binaires

Stéphane Vialette

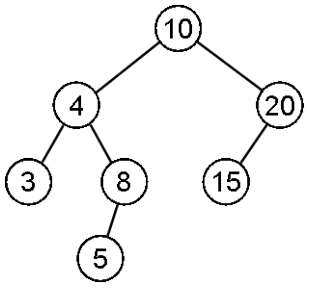
20 octobre 2020

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
...⊥
...⊥
..⊥
```

(b) Écrire la fonction

```
emptyBT :: BTree a
```

qui retourne un arbre binaire vide.

(c) Écrire la fonction

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

qui retourne le nombre de sommets dans un arbre binaire.

(d) Écrire la fonction

```
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.

(e) Écrire la fonction

```
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.

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

(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>
```

(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.

(i) Écrire la fonction

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

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

(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]
```

(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]
```

(l) Écrire la fonction

```
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
[]
```

(m) Écrire la fonction

```
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
...⊥
...⊥
..⊥
```

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).

(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.

(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
..⊥
..⊥
```

(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
...⊥
...⊥
..⊥
```

(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`.

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.

(b) Ecrire la fonction

```
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]
```