Kurs języka Haskell

Lista zadań na pracownię nr 3

Do zgłoszenia w SKOS-ie do 27 marca 2020

W poniższych zadaniach będziemy implementować różne rodzaje binarnych drzew poszukiwań rozważanych na poprzedniej liście. W szczególności będziemy balansować drzewa binarne. Tam, gdzie informacja o zbalansowaniu drzewa nie jest potrzebna, możemy patrzeć na różne rodzaje drzew poprzez *widok*, ujawniający jedynie etykietę korzenia drzewa i jego dwa poddrzewa. Definiujemy w tym celu typ

```
data BTree t a = Node (t a) a (t a) | Leaf
```

Zauważmy, że parametr `t` typu `BTree` ma rodzaj (*kind*) `t :: * -> *`, tj. jest typem polimorficznym z jednym parametrem typowym. Rozważmy klasę

```
class BT t where
   toTree :: t a -> BTree t a
```

Polecenie 1. Przeczytaj podrozdział *11.3.9. View patterns* dokumentacji kompilatora GHC *Glasgow Haskell Compiler User's Guide.*

Jeśli jakiś abstrakcyjny typ drzew (np. implementacja drzew AVL) jest instancją klasy `BT` (od Binary Tree), to metoda `toTree` ujawnia etykietę wierzchołka i poddrzewa tego drzewa. Używając *view patterns* możemy korzystać z dopasowania wzorca, mimo iż nie znamy konstruktorów danego typu drzew. Na przykład

```
is_empty :: BT t => t -> Bool
is_empty (toTree -> Leaf) = True
is_empty (toTree -> Node _ _ _) = False
```

jest *obserwatorem* (tj. operacją, która nie zmienia drzewa, a jedynie odczytuje z niego pewne informacje) sprawdzającym czy drzewo jest puste. Do poprawnego skompilowania powyższy kod wymaga włączenia rozszerzenia języka `ViewPatterns`. Klauzula

```
f (toTree -> Node l x r) = e
```

jest równoważna następującej:

```
f t = case toTree t of
   Node l x r -> e
```

ale znacznie bardziej zwięzła i czytelna.

Rozważmy teraz niezbalansowane drzewa binarne:

```
data UTree a = UNode (UTree a) a (UTree a) | ULeaf
```

Możemy zdefiniować dla drzew tego typu różne obserwatory:

```
treeSize :: UTree a -> Int
treeSize ULeaf = 0
treeSize (UNode l _ r) = treeSize l + treeSize r

treeHeight :: UTree a -> Int
```

```
treeHeight ULeaf = 0
treeHeight (UNode l _ r) = 1 + max (treeHeight l) (treeHeight r)

treeLabels :: UTree a -> [a]
treeLabels = flip aux [] where
   aux ULeaf acc = acc
   aux (UNode l x r) acc = aux l (x : aux r acc)

treeFold :: (b -> a -> b -> b) -> b -> UTree a -> b
treeFold _ e ULeaf = e
treeFold n e (UNode l x r) = n (treeFold n e l) x (treeFold n e r)
```

Zadanie 1 (1 pkt). Uogólnij powyższe operacje na instancje klasy `BT`, tj. zaprogramuj funkcje

```
treeSize :: BT t => t a -> Int
treeLabels :: BT t => t a -> [a]
treeFold :: BT t => (b -> a -> b -> b) -> b -> t a -> b
```

Aby uzyskać dostęp do etykiety wierzchołka i jego poddrzew użyj *view patterns.*

Jeśli teraz zainstalujemy typ `UTree` w klasie `BT`:

```
instance BT UTree where
   toTree ULeaf = Leaf
   toTree (UNode l x r) = Node l x r
```

to powyższe operacje będą dla niego dostępne bez konieczności osobnej implementacji.

Zauważ, że drzewa niezbalansowane możemy też zdefiniować jako punkt stały operatora `BTree`:

```
newtype Unbalanced a = Unbalanced { fromUnbalanced :: BTree Unbalanced a }
```

Instalacja typu `Unbalanced` w klasie `BT` jest niezwykle prosta:

```
instance BT Unbalanced where
   toTree = fromUnbalanced
```

Zadanie 2 (1 pkt). Używając *view patterns* zaimplementuj funkcję

```
searchBT :: (Ord a, BT t) => a -> t a -> Maybe a
```

która ujawnia etykietę wierzchołka drzewa równą podanej lub zwraca `Nothing` w razie, gdy takiej etykiety nie ma. Ponieważ metoda `compare` klasy `Ord` jest jedynie praporządkiem, to ważne jest, aby zwrócić etykietę wierzchołka, a nie element podany jako argument. Zaimplementuj też proste funkcje

```
toUTree :: BT t => t a -> UTree a
toUnbalanced :: BT t => t a -> Unbalanced a
```

rzutujące dowolne drzewa binarne na konkretne typy drzew niezbalansowanych.

Zadanie 3 (1 pkt). Drzewa binarne o etykietowanych wierzchołkach wewnętrznych możemy zwięźle przedstawiać w postaci napisów następująco: dywiz `-` oznacza drzewo puste. Drzewo niepuste przedstawiamy w postaci napisu $l\textvisiblespace x\textvisiblespace r$, gdzie $\textvisiblespace$ oznacza znak spacji, x jest napisem reprezentującym etykietę wierzchołka (otrzymanym za pomocą metody `show` klasy `Show`), zaś l i r są napisami reprezentującymi lewe i prawe poddrzewo ujętymi w nawiasy okrągłe z wyjątkiem przypadku, gdy reprezentują drzewo puste. Na przykład:

Wartość typu `UTree`	Odpowiadający jej napis
`ULeaf`	`"-"`
`UNode ULeaf 13 ULeaf`	`"- 13 -"`
`UNode (UNode ULeaf 1 (UNode ULeaf 2 ULeaf)) 3 ULeaf`	`"(- 1 (- 2 -)) 3 -"`

```
   ┌15
  ┌14
  │└13
 ┌12
 ││┌11
 │└10
 │ └9
─8
 │ ┌7
 │┌6
 ││└5
 └4
  │┌3
  └2
   └1
```

Rysunek 1: Przedstawienie drzewa binarnego w postaci *ascii art* z wykorzystaniem symboli *box drawing*

Uzupełnij poniższą deklarację instancji klasy `Show`:

```
instance (BT t, Show a) => Show (t a) where
   show = ...
```

Dla poprawnej kompilacji jest potrzebne rozszerzenie języka `FlexibleInstances`. Zauważ, że nie instalujesz w klasie `Show` pojedynczego typu, ale na raz wszystkie typy będące instancjami klasy `BT`.

Powyższe zadanie nieco nadużywa klasy `Show`. Zwykle przyjmuje się, że metoda `show` powinna utworzyć napis, który jest poprawnym wyrażeniem języka, którego wartością jest argument tej metody.

Zadanie 4 (2 pkt). Wiele programów konsolowych, np. `pstree`, wypisuje drzewa używając znaków Unicode ┌ (`U+250C`), └ (`U+2514`), │ (`U+2502`) i ─ (`U+2500`). Drzewo jest rysowane w pozycji obróconej o 90° przeciwnie do ruchu wskazówek zegara: korzeń znajduje się przy lewej krawędzi ekranu, prawe poddrzewo jest na górze, a lewe — na dole ekranu. Etykieta każdego wierzchołka jest zapisana w osobnym wierszu i rozpoczyna się w kolumnie $k+2$, gdzie k jest długością ścieżki w drzewie od korzenia do wierzchołka zawierającego tę etykietę. Na przykład pełne uporządkowane drzewo binarne o wierzchołkach etykietowanych liczbami od 1 do 15 jest przedstawione na Rysunku 1. Zaprogramuj funkcję

```
treeBoxDrawing :: (BT t, Show a) => t a -> String
```

przedstawiającą drzewo w tej reprezentacji. Podobnie jak w poprzednim zadaniu do przedstawienia etykiet drzewa użyj metody `show`.

Zadanie 5 (bonus 2 pkt). Jeszcze ładniejszy sposób graficznego przdstawiania drzew, to H drzewa (zob. hasło *H tree* w anglojęzycznej Wikipedii). Zaprogramuj funkcję

```
htree :: (BT t, Show a) => t a -> String
```

przedstawiającą drzewo w tej reprezentacji. Podobnie jak w poprzednim zadaniu do przedstawienia etykiet drzewa użyj metody `show`.

Aby móc nie tylko obserwować abstrakcyjne drzewa binarne, ale także jest przetwarzać (tj. definiować *operatory* działające na drzewach) potrzebujemy poza abstrakcyjnym *destruktorem* `toTree` także abstrakcyjne *konstruktory*. W tym celu definiujemy klasę

```
class BT t => BST t where
   node :: t a -> a -> t a -> t a
   leaf :: t a
```

Metody tej klasy są niekiedy nazywane *smart konstruktorami*, gdyż poza alokacją pamięci dla nowego wierzchołka drzewa potrafią jeszcze wykonać pewne obliczenia (np. zbalansować drzewo).

Ponieważ konstruktory abstrakcyjne drzew niezbalansowanych są *dumb*, a nie *smart* (tj. nie wykonują żadnego przebudowania drzewa), to instalacja typów `UTree` i `Unbalanced` w klasie `BST` jest bardzo prosta:

```
instance BST UTree where
   node = UNode
   leaf = ULeaf
instance BST Unbalanced where
   node l x r = Unbalanced $ Leaf l x r
   leaf = Unbalanced Leaf
```

Abstrakcyjne smart konstruktory `node` i `leaf` wraz z destruktorem `toTree` (używanym wygodnie we *view patternach*) pozwalają zaimplementować operacje na drzewach binarnych poszukiwań, które dają im funkcjonalność zbiorów elementów:

```
class Set s where
   empty  :: s a
   search :: Ord a => a -> s a -> Maybe a
   insert :: Ord a => a -> s a -> s a
   delMax :: Ord a => s a -> Maybe (a, s a)
   delete :: Ord a => a -> s a -> s a
```

Zadanie 6 (2 pkt). Dokończ poniższą deklarację instancji:

```
instance BST s => Set s where
   ...
```

Zaimplementuj przy tym algorytmy wstawiania i usuwania elementów na wzór algorytmów dla drzew niezbalansowanych. Zauważ, że metoda `search` została już wcześniej zdefiniowana dla dowolnej instancji klasy `BT`, a klasa `BST` jest podklasą tej klasy. Aby kompilacja przebiegła poprawnie musisz włączyć rozszerzenie języka `UndecidableInstances`.

Zauważ, że po rozwiązaniu powyższego zadania mamy już metody klasy `Set` dostępne dla typów `UTree` i `Unbalanced`. Naszym zadaniem jest teraz zdefiniowanie innych rodzajów drzew — drzew zbalansowanych i zainstalowanie ich w klasie `BST` (a zatem też w klasie `BT`). Dla tych drzew zaprogramujemy jedynie operacje balansowania w postaci metody `node`. Dzięki rozwiązaniu zadania 6 nie będziemy musieli programować dla nich osobno operacji będących metodami klasy `Set`.

Drzewa binarne *zbalansowane względem rozmiaru* (zwane też *drzewami Adamsa*), to drzewa, które w każdym wierzchołku przechowują liczbę wierzchołków poddrzewa, którego ten wierzchołek jest korzeniem:

```
data WBTree a = WBNode (WBTree a) a Int (WBTree a) | WBLeaf
```

Oczywiście funkcja `treeSize` z zadania 1 potrafi obliczyć rozmiar również takich drzew, ale potrzebujemy ujawniać rozmiar drzew w czasie stałym:

```
wbsize :: WBTree a -> Int
wbsize (WBNode _ _ n _) = n
wbsize WBLeaf = 0
```

Drzewo jest ω-*zbalansowane w korzeniu*, gdzie $\omega > 1$, jeśli:

- jest liściem `WBLeaf` lub
- jest postaci `WBNode` $l\ x\ n\ r$ oraz
  - `wbsize` $l \le 1$ i `wbsize` $r \le 1$ lub

– `wbsize` $l \leq \omega \cdot$ `wbsize` r i `wbsize` $r \leq \omega \cdot$ `wbsize` l.

Drzewo jest *ω-zbalansowane względem rozmiaru*, jeśli każde jego poddrzewo jest ω-zbalansowane w korzeniu.

Polecenie 2. Udowodnij, że drzewa ω-zbalansowane względem rozmiaru są zbalansowane, tj. dla każdego takiego drzewa t jest `treeHeight` $t = O(\log($`wbsize` $t))$.

Jeśli drzewo uległo rozbalansowaniu, to możemy przywrócić niezmiennik zbalansowania za pomocą rotacji.

Polecenie 3. Przeczytaj artykuł: Stephen Adams, Efficient sets — a balancing act, *J. Funct. Program.* **3**(4):553–561, 1993. Kopia jest dostępna na stronie zajęć. Jeśli artykuł wyda Ci się zbyt krótki i lakoniczny, to zajrzyj do jego rozszerzonej wersji: Stephen Adams, Implementing Sets Efficiently in a Functional Language, *Tech. Report* CSTR 92–10, University of Southampton, 1992.

Zadanie 7 (2 pkt). Zainstaluj typ `WBTree` w klasach `BT` i `BST`. Jedyną nietrywialną funkcją do zaprogramowania jest metoda `node`, w której trzeba zaimplementować cztery rotacje drzew opisane w artykule Stephena Adamsa. Przyjmij $\omega = 5$.

Zadanie 8 (2 pkt). Drzewa binarne *zbalansowane względem wysokości* są podobne do drzew zbalansowanych względem rozmiaru, jednak zamiast rozmiaru przechowują w każdym wierzchołku wysokość poddrzewa, którego ten wierzchołek jest korzeniem:

```
data HBTree a = HBNode (HBTree a) a Int (HBTree a) | HBLeaf
```

Zamiast ogólnej funkcji `treeHeight` z zadania 1 działającej w czasie liniowym względem rozmiaru drzewa mamy teraz funkcję działającą w czasie stałym:

```
hbheight :: WBTree a -> Int
hbheight (WBNode _ _ h _) = h
hbheight WBLeaf = 0
```

Niezmienniki tych drzew też są podobne: drzewo jest δ-zbalansowane w korzeniu, jeśli

- jest liściem `HBLeaf` lub
- jest postaci `HBNode` $l\ x\ h\ r$ oraz
  - `hbheight` $l \leq 1$ i `hbheight` $r \leq 1$ lub
  - `hbheight` $l \leq$ `hbheight` $r + \delta$ i `hbheight` $r \leq$ `hbheight` $l + \delta$.

Zauważ, że AVL-drzewa są bardzo podobne do drzew zbalansowanych względem wysokości, w wierzchołkach przechowują jednak *różnice wysokości poddrzew* a nie wysokości drzew, co pozwala zmniejszyć liczbę bitów dodatkowej informacji do dwóch. Rotacje przywracające zbalansowanie są jednak bardzo podobne.

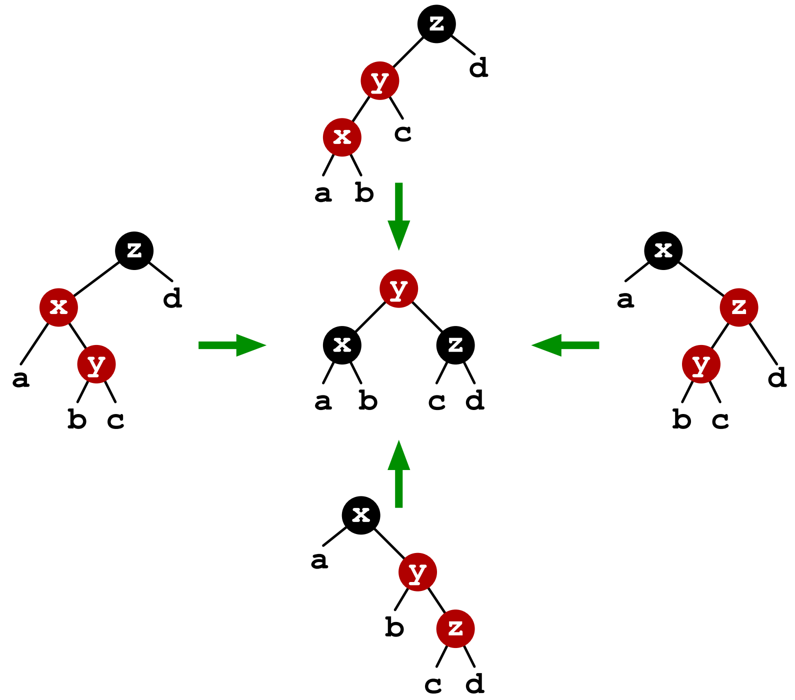
Zainstaluj drzewa zbalansowane względem wysokości w klasach `BT` i `BST`. Przyjmij $\delta = 1$ (tak jak w drzewach AVL).

Znane i lubiane drzewa *czerwono-czarne*, to drzewa, które w każdym wierzchołku przechowują tylko jeden dodatkowy bit informacji — kolor wierzchołka — czerwony lub czarny:

```
data Color = Red | Black
data RBTree a = RBNode (RBTree a) a Color (RBTree a) | RBLeaf
```

Dobrze znane niezmienniki są następujące:

- czerwony wierzchołek ma czarnego ojca,
- na każdej ścieżce od korzenia do liścia jest tyle samo czarnych wierzchołków.



Rysunek 2: Rotacje drzew czerwono-czarnych

Podobnie jak w przypadku poprzednich typów drzew zaburzenie niezmiennika spowodowane wstawieniem bądź usunięciem elementu można naprawić poprzez rotacje przedstawione na Rysunku 2.

Polecenie 4. Przeczytaj pracę: Chris Okasaki, Red-black trees in a functional setting, *J. Funct. Program.* **9**(4):471–477, 1999.

Niestety próba zainstalowania drzew czerwono-czarnych w klasie `BST`, tj. próba zaimplementowania rotacji w postaci *smart konstruktora* `node` i otrzymania za darmo zestawu funkcji `insert`, `delMax` i `delete` dla tych drzew autorowi tej listy zadań się nie udała. Nie potrafi on też tak zrefaktorować definicji klasy `BST`, aby było to możliwe. To samo dotyczy także innych rodzajów drzew, takich jak drzewa AVL. Ostatnie zadanie pokazuje zatem ograniczenia obiecującej metody *smart konstruktorów*, w której rozdzieliliśmy konkretną implementację drzew zbalansowanych od ich abstrakcyjnego interfejsu.

Zadanie 9 (2 pkt). Zainstaluj drzewa czerwono-czarne w klasie `BT` (a zatem `search` już mamy!). Spróbuj zainstalować drzewa czerwono-czarne w klasie `BST`. Jeśli Ci się to nie uda, to zainstaluj te drzewa wprost w klasie `Set`.

Najbardziej denerwujące jest to, że kod metod `insert`, `delMax` i `delete` jest irytująco podobny do tego, który napisaliśmy w zadaniu 6, a jednak autor listy musiał go napisać ponownie. Czy potrafisz tak zrefaktorować kod z bieżącej listy, aby powtórna implementacja nie była konieczna?