Programowanie Funkcyjne 2018

Lista zadań nr 10

na 19 grudnia 2018

Zadanie 1 (4 pkt). Przypomnijmy typ danych reprezentujący drzewa binarne etykietowane w węzłach:

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

- Zdefiniuj funkcję `dfnum :: BTree a -> BTree Integer` numerującą węzły drzewa binarnego w kolejności przechodzenia wgłąb. Przykładowo, ponumerowaną wersją drzewa

  ```
  Node (Node (Node Leaf 'a' Leaf) 'b' Leaf) 'c' (Node Leaf 'd' Leaf)
  ```

  jest drzewo

  ```
  Node (Node (Node Leaf  3  Leaf)  2  Leaf)  1  (Node Leaf  4  Leaf)
  ```

- Zdefiniuj funkcję `bfnum :: BTree a -> BTree Integer` numerującą węzły drzewa w kolejności przechodzenia go wszerz. Tak ponumerowaną wersją drzewa z poprzedniego przykładu będzie

  ```
  Node (Node (Node Leaf  4  Leaf)  2  Leaf)  1  (Node Leaf  3  Leaf)
  ```

Wskazówka: lasy numeruje się łatwiej niż drzewa.

Zadanie 2 (4 pkt). Tablica funkcyjna to struktura danych, która podobnie jak tablica imperatywna, pozwala na swobodny dostęp do swoich składowych (poprzez ich indeksy w tablicy). Jednakże, w przeciwieństwie do tablicy imperatywnej, operacje modyfikujące składowe tablicy funkcyjnej nie nadpisują istniejącej tablicy, a tworzą jej kopię, przy czym oryginalna kopia nadal istnieje i może być używana w dalszych obliczeniach. Takie struktury sprawdzają się lepiej niż tablice imperatywne np. w algorytmach niedeterministycznych z nawrotami.

Rozważmy implementację tablic funkcyjnych za pomocą drzew binarnych z poprzedniego zadania.

Zakładamy przy tym, że drzewo reprezentuje tablicę indeksowaną liczbami całkowitymi od 1 do n, a ścieżka do składowej o indeksie k, wyznaczona jest przez serię dzieleń modulo 2, aż do osiagnięcia wartości 1, wg zasady: jeśli $k \mod 2 = 0$, to wybieramy lewego syna, a w przeciwnym razie - prawego, a następnie poszukujemy elementu o indeksie $k\,\mathrm{div}\,2$. W przypadku drzew zbalansowanych, a z takimi mamy tu do czynienia, dostęp do k-tego elementu wymaga $\log k$ kroków.

Zdefiniuj typ danych `Array a` (przechowujący, oprócz drzewa, metadane usprawniające działanie operacji) oraz następujące operacje na tablicach funkcyjnych:

- `aempty :: Array a`, tablica pusta;
- `asub :: Array a -> Integer -> a`, pobranie składowej o zadanym indeksie;
- `aupdate :: Array a -> Integer -> a -> Array a`, modyfikacja składowej o zadanym indeksie;
- `ahiext :: Array a -> a -> Array a`, rozszerzenie tablicy o jedną składową;
- `ahirem :: Array a -> Array a`, usunięcie składowej o najwyższym indeksie.

Zadanie 3 (4 pkt). Chcemy zdefiniować funkcję sprintf znaną z języka C, tak by np.

```
sprintf "Ala ma %d kot%s." :: Integer -> String -> String
```

pozwalało zdefiniować funkcję

```
\ n -> sprintf "Ala ma %d kot%s." n
  (if n = 1 then "a" else if 1 < n & n < 5 then "y" else "ow")
```

Na pierwszy rzut oka wydaje się, że rozwiązanie tego zadania wymaga typów zależnych, ponieważ typ funkcji `sprintf` zależy od jej pierwszego argumentu. Okazuje się jednak, że polimorfizm parametryczny wystarczy. Dla uproszczenia załóżmy, że format nie jest zadany przez wartość typu string (nie chcemy zajmować się parsowaniem), ale przez konkatenację następujących dyrektyw formatujących:

- `lit s` - stała napisowa s
- `eol` - koniec wiersza
- `int` - liczba typu `Integer`
- `flt` - liczba typu `Float`
- `str` - napis typu `String`

Zakładając, że operatorem konkatenacji dyrektyw jest `^^`, powyższy przykład może być zapisany następująco:

```
sprintf (lit "Ala ma " ^^ int ^^ lit " kot" ^^ str ^^ lit ".")
```

Zdefiniuj funkcje `lit`, `eol`, `inr`, `flt`, `str`, `^^` oraz funkcję `sprintf`. Nie należy używać klas typów!
Wskazówka: dyrektywy powinny być funkcjami transformującymi kontynuacje, a operator `^^` to zwyczajne złożenie takich funkcji. Na przykład `int` powinien mieć typ `(String -> a) -> String -> (Integer -> a)` (argumentem ma być kontynuacja oczekująca napisu, ale o nieokreślonym typie odpowiedzi, a wynikiem ma być kontynuacja oczekująca napisu, a następnie liczby całkowitej). Podobnie, typem `eol` będzie `(String -> a) -> String -> a`.

Zadanie 4 (4 pkt). Drzewa czerwono-czarne to jeden z rodzajów drzew BST które zawsze są zbalansowane. Oprócz standardowych informacji, wszystkie wierzchołki wewnętrzne przechowują dodatkową, kolor, a na drzewa nałożone są pewne niezmienniki:

- korzeń niepustego drzewa jest zawsze czarny
- czarna wysokość (tj. liczba czarnych wierzchołków) każdych dwóch ścieżek w drzewie jest taka sama
- żaden z synów czerwonego wierzchołka nie może być czerwony

Powyższe niezmienniki można wyrazić w Haskellu za pomocą zaawansowanych aspektów systemu typów, ale my nie będziemy (na razie) tego robić. Zamiast tego zdefiniujemy drzewa czerwono czarne następująco:

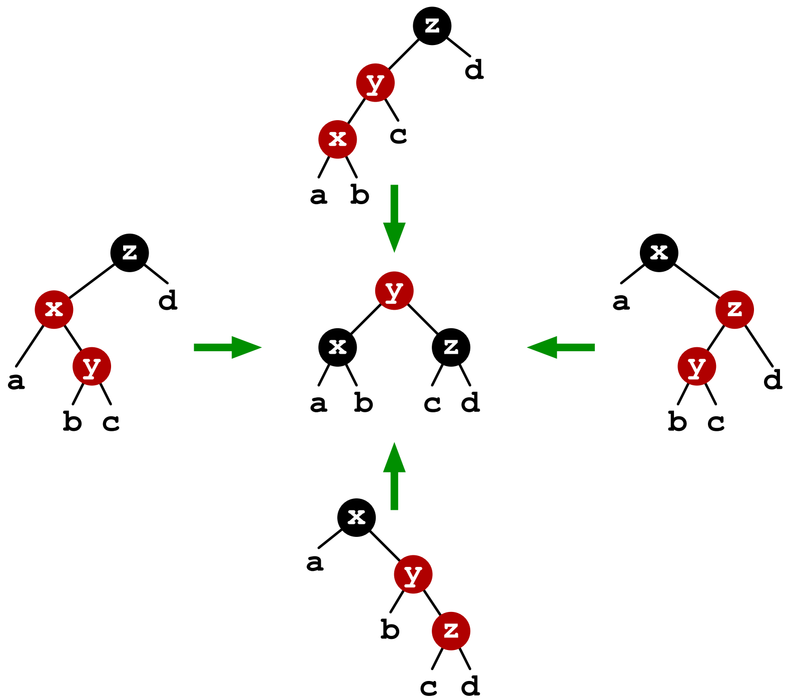
```
data Color = Red | Black
data RBTree a = RBNode Color (Tree a) a (Tree a) | RBLeaf
```

Wstawianie elementów do drzew czerwono-czarnych jest stosunkowo proste: jeśli nowo utworzony węzeł pokolorujemy na czerwono, to jedynym niezmiennikiem który możemy zepsuć jest ostatni: nowy węzeł może być synem czerwonego węzła. Dlatego, odtwarzając ścieżkę do korzenia musimy naprawić nasz niezmiennik. Szczęśliwie, można to zrobić w prosty sposób wykorzystując operacje (nazywane zazwyczaj „rotacjami") przedstawione na Rys. 1.

W Haskellu operacje te możemy zaimplementować jako *smart constructor* `rbnode` o takim samym typie jak zwykły konstruktor `RBNode`. Implementacja konstruktora `rbnode` polega na sprawdzeniu czy skonstruowalibyśmy jedno ze „złych" drzew na rysunku – i jeśli zachodzi taki przypadek, to zastąpienie go naprawionym drzewem na środku. (Zauważ że rotacje naprawiają niezmienniki w rozważanym poddrzewie, ale otrzymany korzeń jest czerwony, więc może naruszać trzeci niezmiennik bliżej korzenia.)

Po odbudowaniu całej ścieżki jedynym miejscem gdzie możemy naruszyć niezmiennik jest korzeń (może być czerwonym węzłem z czerwonym dzieckiem) – ale korzeń zawsze możemy pokolorować na czarno (dlaczego?).

Zaimplementuj *smart constructor* `rbnode :: Color -> RBTree a -> a -> RBTree a -> RBTree a` oraz operację wstawiania elementu do drzewa `rbinsert :: a -> RBTree a -> RBTree a`.



Rysunek 1: Rotacje przy wstawianiu do drzewa czerwono-czarnego.

Zadanie 5 (4 pkt). Zaimplementuj funkcję `rbtreeFromList : [a] -> RBTree a` która buduje drzewo czerwono-czarne zawierające elementy podanej *posortowanej* listy. Czas działania tej funkcji powinien być liniowy względem długości listy, a niezmienniki drzewa czerwono-czarnego powinny być zachowane.
Wskazówka: zacznij od zastanowienia się jak w czasie liniowym stworzyć maksymalnie zbalansowane drzewo binarne zawierające elementy listy w zadanej kolejności. Następnie zastanów się jaki kolor powinny mieć poszczególne wierzchołki (Twoja implementacja nadal powinna wykonywać tylko jeden przebieg, ale warto rozbić proces projektowania na dwa podproblemy).