Programowanie Funkcyjne 2018

Lista zadań nr 9: Metody implementacji struktur danych w Haskellu

Na zajęcia 20 grudnia 2018

W zadaniach z bieżącej listy będziemy używać różnych rozszerzeń standardu Haskell'98. Aby zachować zgodność ze standardem, kompilator ghc umożliwia korzystanie z tych rozszerzeń tylko wtedy, gdy jawnie tego zażądamy, umieszczając na początku pliku źródłowego odpowiednią *pragmę*. Pragma, to rodzaj komentarza zrozumiałego dla kompilatora. Aby poprawnie skompilować znajdujące się poniżej deklaracje należy na początku pliku źródłowego (przed pierwszą deklaracją) umieścić następujący wiersz:

```
{-# LANGUAGE KindSignatures, MultiParamTypeClasses, FlexibleInstances #-}
```

Standardowe biblioteki Haskella są w miarę kompletne i dobrze przemyślane. Jednym z dotkliwych niedopatrzeń jest brak funkcji:

```
(><) :: (a -> b) -> (a -> c) -> a -> (b,c)
(f >< g) x = (f x, g x)
```

która mogłaby się znaleźć np. w module `Data.Tuple`. Kombinator

```
warbler :: (a -> a -> b) -> a -> b
warbler f x = f x x
```

jest co prawda zdefiniowany w module `Data.Aviary.Birds`, ale moduł ten nie jest dostępny w standardowym środowisku ghc i wymaga doinstalowania. Przyda nam się też standardowy anamorfizm dla list i katamorfizm dla wartości logicznych. W tym celu na początku pliku należy umieścić dyrektywy:

```
import Data.List (unfoldr)
import Data.Bool (bool)
```

W kilku poniższych zadaniach będziemy implementować kolejki priorytetowe, opisane następującą specyfikacją, w której metoda `single` tworzy kolejkę jednoelementową zawierającą podany element:

```
class Ord a => Prioq (t :: * -> *) (a :: *) where
   empty      :: t a
   isEmpty    :: t a -> Bool
   single     :: a -> t a
   insert     :: a -> t a -> t a
   merge      :: t a -> t a -> t a
   extractMin :: t a -> (a, t a)
   findMin    :: t a -> a
   deleteMin  :: t a -> t a
   fromList   :: [a] -> t a
   toList     :: t a -> [a]
   insert = merge . single
   single =  flip insert empty
   extractMin = findMin >< deleteMin
   findMin = fst . extractMin
   deleteMin = snd . extractMin
   fromList = foldr insert empty
   toList = unfoldr . warbler $ bool (Just . extractMin) (const Nothing) . isEmpty
```

Dla niektórych funkcji podano domyślne implementacje. W każdej instancji należy zdefiniować co najmniej jedną z funkcji `insert` i `single` oraz funkcję `extractMin` lub obie funkcje `findMin` i `deleteMin`. Pozostałe funkcje będą miały domyślną implementację, choć *możemy* zadeklarować własną, być może bardziej efektywną. Domyślna implementacja ostatniej metody mogłaby właściwie wyglądać tak:

```
toList = unfoldr (\ t -> if isEmpty t then Nothing else Just (extractMin t))
```

ale ze względów estetycznych wolimy oryginalną.

Zadanie 1 (1 pkt). Zaprogramuj funkcję

```
(<+>) :: Ord a => [a] -> [a] -> [a]
```

scalającą dwie uporządkowane listy w jedną.

Zadanie 2 (2 pkt). Rozważmy implementację kolejki priorytetowej w postaci uporządkowanej listy elementów:

```
newtype ListPrioq a = LP { unLP :: [a] }
```

Zainstaluj typ `ListPrioq` w klasie `Prioq`.

Zadanie 3 (2 pkt). Zaprogramuj w Ocamlu sygnatury typów uporządkowanych `ORDERED` i kolejek priorytetowych `PRIOQ`. Zaprogramuj funktor, który przyporządkowuje implementacji typu uporządkowanego implementację kolejek priorytetowych wykorzystującą do przechowywania elementów, tak jak w poprzednim zadaniu, uporządkowane listy.

Zadanie 4 (2 pkt). Efektywne implementacje kolejek priorytetowych wykorzystują drzewa (binarne lub wieloarne) spełniające *warunek kopca*: w każdym poddrzewie etykieta korzenia jest nie większa niż etykiety synów tego drzewa. Najmniejszy element drzewa spełniającego warunek kopca znajduje się zawsze w jego korzeniu.

Potrzebujemy drzew spełniających warunek kopca które, dla pewnego k, zawierają dokładnie 2^k elementów. Liczba k nazywa się *stopniem* drzewa. Jednymi z bardziej popularnych drzew tego rodzaju są drzewa *dwumianowe*. Są to drzewa wieloarne zdefiniowane za pomocą następującej definicji rekurencyjnej:

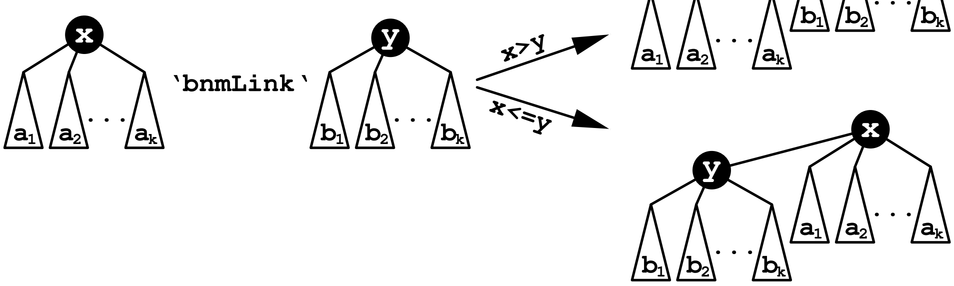
- Drzewo stopnia 0 zawiera jeden wierzchołek.
- Drzewo stopnia $k+1$ zawiera korzeń posiadający $k+1$ synów będących drzewami stopnia od 0 do k w kolejności malejącej.

W Haskellu drzewa dwumianowe możemy reprezentować następująco:

```
data BinomialTree a = Binom a [BinomialTree a]
```

Niestety jest wiele wartości typu `BinomialTree`, które nie są poprawnymi drzewami dwumianowymi. Poprawimy to na kolejnych zajęciach, a na razie będziemy z tym jakoś żyć.

Będziemy używać drzew dwumianowych spełniających warunek kopca. Musimy umieć łączyć dwa drzewa tego samego stopnia k w jedno drzewo stopnia $k+1$ tak, by zachować warunek kopca. Nie jest to trudne. Szczegóły wyjaśnia poniższy obrazek:



Zaprogramuj funkcje:

```
bnmLink :: Ord a => BinomialTree a -> BinomialTree a -> BinomialTree a
bnmUnlink :: Ord a => BinomialTree a -> (a, [BinomialTree a])
```

Druga z nich ujawnia korzeń drzewa dwumianowego i listę jego synów w kolejności *rosnących* stopni.

W niektórych zastosowaniach potrzebujemy poznać stopień drzewa w czasie $O(1)$, zatem sprawdzenie długości listy jego synów jest zbyt kosztowne. Możemy opakować dowolny rodzaj drzew które mają atrybut stopnia za pomocą typu

```
data Sized (t :: * -> *) (a :: *) = Sized Int (t a)
```

Drzewo typu `Sized BinomialTree a` jest w istocie parą złożoną z liczby całkowitej reprezentującej stopień drzewa i drzewa właściwego. Zaprogramuj funkcje:

```
sbnmLink :: Ord a => Sized BinomialTree a -> Sized BinomialTree a
                       -> Sized BinomialTree a
sbnmUnlink :: Ord a => Sized BinomialTree a -> (a, [Sized BinomialTree a])
```

Funkcja `sbnmLink` w odróżnieniu od `bnmLink` nie wierzy na słowo, że jej argumenty są tego samego stopnia, lecz może to sprawdzić i przerywa działanie programu w razie błędu.

Zadanie 5 (3 pkt). *Rzadkie kopce dwumianowe* to listy drzew dwumianowych spełniających warunek kopca uporządkowane według rosnących stopni, w których nie ma dwóch drzew tego samego stopnia. Możemy je przedstawić w Haskellu za pomocą typu

```
newtype SparseBinomialHeap a = SBH [Sized BinomialTree a]
```

Aby scalić dwa kopce, należy złączać odpowiadające sobie drzewa tego samego stopnia, podobnie jak dodaje się liczby w zapisie binarnym. Aby znaleźć element najmniejszy, należy porównać elementy znajdujące się w korzeniach drzew i wybrać najmniejszy z nich. Aby usunąć element najmniejszy, należy wyszukać drzewo zawierające ten element, usunąć je z kopca, skorzystać z operacji `unlink`, potraktować otrzymaną listę drzew jako kopiec i scalić go z resztą oryginalnego kopca. Zainstaluj typ `SparseBinomialHeap` w klasie `Prioq`.

Zadanie 6 (3 pkt). *Gęste kopce dwumianowe* to listy których k-ty element (numerujemy od zera) albo jest drzewem dwumianowym stopnia k spełniającym warunek kopca, albo zawiera informację, że drzewo stopnia k nie występuje w kopcu. W takich kopcach możemy wykorzystać drzewa dwumianowe nie zawierające informacji o swoim stopniu, gdyż wynika ona z położenia danego drzewa na liście:

```
newtype DenseBinomialHeap a = DBH [Maybe (BinomialTree a)]
```

Zainstaluj typ `DenseBinomialHeap` w klasie `Prioq`.

Zadanie 7 (3 pkt). Wadą drzew dwumianowych jest to, że są wieloarne, przez co gospodarują pamięcią dosyć rozrzutnie. Zauważmy, że idealnym kandydatem na drzewa w implementacjach kopców byłyby pełne drzewa binarne, ale zawierają one jedynie $2^k - 1$ wierzchołków. *Proporczyki* stopnia k to drzewa, których korzeń ma jednego syna, który jest pełnym drzewem binarnym o wysokości k:

```
data BinTree a = Leaf | Node (BinTree a) a (BinTree a)
data Pennant a = Pennant a (BinTree a)
```

Podobnie jak w przypadku drzew dwumianowych definicja typu `BinTree` jest zbyt ogólna i dopuszcza drzewa, które nie są pełne. W kolejnych tygodniach to poprawimy. Zaprogramuj operacje

```
pennantLink :: Ord a => Pennant a -> Pennant a -> Pennant a
pennantUnlink :: Ord a => Pennant a -> (a, [Pennant a])
```

które zachowują warunek kopca w przetwarzanych drzewach. Pierwsza łączy dwa proporczyki w jeden stopnia o jeden wyższego. Druga z nich rozbija proporczyk stopnia k na korzeń i listę k proporczyków stopnia od 0 do $k-1$.

Podobnie jak w przypadku drzew dwumianowych chcemy też mieć wersję drzew, które znają swój stopień. Zaprogramuj funkcje:

```
sPennantLink :: Ord a => Sized Pennant a -> Sized Pennant a -> Sized Pennant a
sPennantUnlink :: Ord a => Sized Pennant a -> (a, [Sized Pennant a])
```

Zadanie 8 (2 pkt). *Rzadkie kopce proporczykowe* są podobne do dwumianowych:

```
newtype SparsePennantHeap a = SPH [Sized Pennant a]
```

Zainstaluj typ `SparsePennantHeap` w klasie `Prioq`.

Zadanie 9 (2 pkt). *Geste kopce proporczykowe* są podobne do dwumianowych:

```
newtype DensePennantHeap a = DPH [Maybe (Pennant a)]
```

Zainstaluj typ `DensePennantHeap` w klasie `Prioq`.

Wersja z poprawkami z 15 grudnia 2018 15:36.