Programowanie Funkcyjne 2018

Lista zadań nr 11

9 stycznia 2019

Zadanie 1 (4 pkt). Rozważmy funkcję, która generuje listę na podstawie pewnego stanu początkowego i funkcji generującej kolejną wartość – taką funkcję nazywamy anamorfizmem:

```
ana :: (b -> Maybe (a, b)) -> b -> [a]
ana f st = case f st of
  Nothing               -> []
  Just (v, st') -> v : ana f st'
```

Pokaż, że funkcje `zip`, `iterate`, `map` są anamorfizmami (tj. zdefiniuj je za pomocą funkcji `ana`).

Dualnie, katamorfizm jest funkcją, która konsumuje wartości danego typu. Dla list można ją zdefiniować tak (znasz tę funkcję pod inną nazwą):

```
cata :: (a -> b -> b) -> b -> [a] -> b
cata f v [] = v
cata f v (x:xs) = f x (cata f v xs)
```

Pokaż, że funkcje `length`, `filter` i `map` są katamorfizmami.

Pojęcie anamorfizmu i katamorfizmu można uogólnić i zdefiniować je dla różnych typów danych. Rozważmy teraz polimorficzny typ danych reprezentujących proste wyrażenia arytmetyczne ze zmiennymi:

```
data Expr a b =
    Number b
  | Var a
  | Plus (Expr a b) (Expr a b)
```

Zdefiniuj anamorfizm i katamorfizm dla tego typu, a następnie wykorzystaj katamorfizm do napisania interpretera wyrażeń arytmetycznych:

```
eval :: Num b => [(a, b)] -> Expr a b -> b
```

Zadanie 2 (6 pkt). W kolejnych dwóch zadaniach 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)
```

Rozważmy następującą specyfikację kolejek priorytetowych:

```
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.

Rozważmy implementację kolejki priorytetowej w postaci uporządkowanej listy elementów:

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

Zainstaluj typ `ListPrioq` w klasie `Prioq`.

Zadanie 3 (6 pkt). Dowiedz się, co to jest polimorfizm wyższego rzędu. Jeśli włączymy w Haskellu rozszerzenie

```
{-# LANGUAGE Rank2Types #-}
```

to na przykład deklaracja

```
omega :: (forall a.a) -> b
omega f = f f
```

daje się poprawnie skompilować – to jest polimorfizm drugiego rzędu. (Jeśli kwantyfikator ogólny nie występuje preneksowo w argumentach funkcji tylko jest głębiej zagnieżdżony, potrzeba włączyć rozszerzenie `RankNTypes`.)
Możemy teraz zdefiniować liczebniki Churcha następująco:

```
newtype Church = Church (forall a. (a -> a) -> (a -> a))
```

Zainstaluj typ `Church` w klasach `Eq`, `Ord`, `Show` i `Num`. W ostatnim przypadku przyjmij, że $n - m = 0$ jeśli $n \leq m$.

Zauważ, że zainstalowanie typu `Church` w klasach `Num` i `Show` pozwala używać liczebników Churcha tak jak zwykłych liczb:

```
*Main> 2 + 3 * 4 :: Church
14
```

Zadanie 4 (4 pkt). Rozwiąż zadanie kontrolne do wykładu 10 (gra w nim).