Programowanie Funkcyjne 2018

Lista zadań nr 9

12 grudnia 2018

Rozbudowana składnia wyrażeń listowych w Haskellu pozwala bardzo zwięźle i czytelnie zapisywać nawet skomplikowane funkcje przetwarzające listy. Poniższe zadania zawierają wiele funkcji, które zaprogramowałeś już w Ocamlu. Staraj się nie sugerować rozwiązaniem Ocamlowym, tylko przemyśl, jakie *nowe* środki wyrazu oferuje Haskell.

Zadanie 1 (2 pkt). Napisz wyrażenie, którego wartością jest taka funkcja typu `[Integer] -> [Integer]` (nazwijmy ją f), że $f\,[n_0, n_1, n_2 \ldots]$ jest listą tych spośród liczb $n_1, n_2, \ldots$, które nie dzielą się przez n_0. Jeśli $l =$ `[2..]`, to mamy

$$\begin{aligned}
l &= [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,\ldots] \\
f(l) &= [3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,\ldots] \\
f(f(l)) &= [5,7,11,13,17,19,23,25,29,31,35,37,41,43,47,49,53,55,59,61,\ldots] \\
f(f(f(l))) &= [7,11,13,17,19,23,29,31,37,41,43,47,49,53,59,61,67,71,73,77,\ldots]
\end{aligned}$$

itd. Zauważ, że głowy tak obliczonych list tworzą listę liczb pierwszych. Wykorzystaj ten pomysł do zdefiniowania listy

```
primes :: [Integer]
```

zawierającej wszystkie liczby pierwsze w kolejności rosnącej. Przydadzą się przy tym standardowe funkcje

```
head :: [a] -> a
map :: (a -> b) -> ([a] -> [b])
iterate :: (a -> a) -> a -> [a]
iterate f x = x : (iterate f (f x))
```

Zauważmy, że powyższy algorytm jest bardzo bliski idei sita Eratostenesa. Musieliśmy jedynie zastąpić nieskończoność aktualną, występującą w oryginalnym sformułowaniu Eratostenesa, nieskończonością potencjalną. Robi to za nas Haskell za pomocą leniwego wartościowania.

Zadanie 2 (2 pkt). Zauważmy, że liczba p jest pierwsza, jeśli nie dzieli się przez żadną taką liczbę pierwszą q, że $q^2 \leq p$ (np. żeby sprawdzić, że liczba 13 jest pierwsza, wystarczy spróbować podzielić ją przez 2 i 3). Zatem listę liczb pierwszych można zdefiniować rekurencyjnie:

$$\texttt{primes'} \quad = \quad [\, p \mid p \in [2..], \forall q \in \texttt{primes'}.(q^2 \leq p \Rightarrow q \nmid p) \,]$$

Niestety rekursja w powyższej zależności nie jest dobrze ufundowana — najmniejsza liczba pierwsza, tj. 2, powinna być jawnie podana:

$$\texttt{primes'} \quad = \quad 2 : [\, p \mid p \in [3..], \forall q \in \texttt{primes'}.(q^2 \leq p \Rightarrow q \nmid p) \,]$$

Wykorzystaj powyższy pomysł do zdefiniowania listy liczb pierwszych

```
primes' :: [Integer]
```

Przydadzą się przy tym funkcje biblioteczne

```
all :: (a -> Bool) -> [a] -> Bool
all p = foldr ((&&) . p) True
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p = foldr (\ x xs -> if p x then x:xs else []) []
```

(Pierwsza z nich ma w Haskellu nieco ogólniejszy typ, co w naszym przypadku nie ma znaczenia.)

Zadanie 3 (2 pkt). Definicje bardzo wielu ciągów można przedstawić w postaci zależności rekurencyjnej. Jeśli np. `fib :: [Integer]` jest ciągiem Fibonacciego, to mamy:

$$\begin{array}{rl}
\texttt{fib} = & [1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ \ldots] \\
& \ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ +\ \ + \\
\texttt{tail fib} = & [1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ 610,\ \ldots] \\
& \ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert\ \ \Vert \\
\texttt{fib} = & 1:1:[2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ 610,\ 987,\ \ldots]
\end{array}$$

Wykorzystaj standardową funkcję

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith (+) (x:xs) (y:ys) = x+y : zipWith (+) xs ys
zipWith _ _ _ = []
```

oraz powyższą zależność rekurencyjną do zdefiniowania ciągu `fib`.

Zadanie 4 (2 pkt). Zaprogramuj w Haskellu funkcje

```
iperm, sperm :: [a] -> [[a]]
```

wyznaczające listy wszystkich permutacji podanej listy. Pierwsza z nich generuje permutacje poprzez wstawianie, druga poprzez wybieranie.

Zadanie 5 (2 pkt). Podlistą listy $[x_1, x_2, \ldots, x_n]$ jest lista $[x_{i_1}, x_{i_2}, \ldots, x_{i_k}]$, gdzie $0 \le k \le n$ oraz $1 \le i_1 < i_2 < \ldots < i_k \le n$. Lista n-elementowa posiada 2^n podlist. Zaprogramuj w Haskellu funkcję

```
sublist :: [a] -> [[a]]
```

wyznaczającą listę wszystkich podlist podanej listy.

Zadanie 6 (4 pkt). Zaprogramuj funkcję

```
(><) :: [a] -> [b] -> [(a,b)]
```

wyznaczającą produkt dwóch list w porządku przekątniowym Cantora, tj.

$$[x_1, x_2, x_3, \ldots] \mathbin{\texttt{><}} [y_1, y_2, y_3, \ldots] \quad = \quad [(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_1, y_3), (x_2, y_2), (x_3, y_1), \ldots]$$

Zauważ, że listy mogą być skończone, bądź nie, np. możemy napisać

```
pairs :: (Integer,Integer)
pairs = [0..] >< [0..]
```

Zadanie 7 (6 pkt). Zbiory elementów uporządkowanego typu reprezentujemy często za pomocą drzew binarnych:

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

Możemy w ten sposób reprezentować jedynie zbiory skończone. Aby rodzina reprezentowalnych podzbiorów była algebrą Boole'a możemy do zbiorów skończonych dodać koskończone, tj. takie, których dopełnienie jest skończone:

```
data Set a = Fin (Tree a) | Cofin (Tree a)
```

Zaprogramuj następujące operacje:

```
setFromList :: Ord a => [a] -> Set a
setEmpty, setFull :: Ord a => Set a
setUnion, setIntersection :: Ord a => Set a -> Set a -> Set a
setComplement :: Ord a => Set a -> Set a
setMember :: Ord a => a -> Set a -> Bool
```

Funkcja `setFromlist` tworzy zbiór skończony zawierający elementy podanej listy. Wartości `setEmpty` i `setFull` reprezentują, odpowiednio, zbiór pusty i zbiór zawierający wszystkie elementy typu `a`.