Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 9

Do zgłoszenia w SKOS-ie do 18 maja 2020

Zadanie 1 (3 pkt). Swojego czasu na forum omawialiśmy algorytm generowania n-tej liczby pierwszej:

```
nthPrime :: Int -> Int
nthPrime x = big !! x
 where
  isPrime n   = all (\k -> n `mod` k /= 0) $ takeWhile (\k -> k*k <= n) small
  small       = 2 : [n | n <- [3,5..], isPrime n]
  big         = 2 : [n | n <- [3,5..], isPrime n]
```

Można też ręcznie zrobić fuzję `big` i `!!` i dostać taki algorytm:

```
nthPrime :: Int -> Int
nthPrime x = startFrom 2 x
 where
  isPrime n   = all (\k -> n `mod` k /= 0) $ takeWhile (\k -> k*k <= n) small
  small       = 2 : [n | n <- [3,5..], isPrime n]
  startFrom n 0 | isPrime n = n
                | otherwise = startFrom (n+1) 0
  startFrom n k = startFrom (n+1) $ if isPrime n then k-1 else k
```

Użyj funkcji `par` zdefiniowanej w `Control.Parallel` żeby ten algorytm zrównoleglić. Oczywiście nie wystarczy dopisać `` `par` `` w odpowiednich miejscach, trzeba trochę kod poprzestawiać tak, by móc podzielić listę `big` na odpowiednie paczki, które można obliczać równolegle.

Wskazówka: Podaliśmy wersję po fuzji, bo autorowi zadania wydaje się, że ta wersja jest łatwiejsza do zrównoleglenia. Autor uzyskał następujące rezultaty obliczenia dwumilionowej (licząc od 0) liczby pierwszej:

```
$ ghc --make Test -threaded -O2 -o prime
[1 of 1] Compiling Main             ( Test.hs, Test.o )
Linking prime ...

$ time ./prime
32452867
real 0m19,954s
user 0m19,943s
sys 0m0,004s

$ time ./prime +RTS -N4
32452867
real 0m9,458s
user 0m25,270s
sys 0m0,012s
```

Dzięki czterem rdzeniom udało się uzyskać wynik dwa razy szybciej.

Zadanie 2 (2 pkt). Zdefiniuj funkcję `bucketSort :: [(Int, a)] -> [(Int, a)]` implementującą kubełkowe sortowanie listy asocjacyjnej, gdzie klucze są typu `Int`. Użyj mutowalnej tablicy w monadzie ST (gdzie wartościami w tablicy są np. listy wartości).

Porównaj prędkość działania tego rozwiązania z rozwiązaniem używającym bibliotecznej funkcji `sort`. Pamiętaj porównywać programy skompilowane z opcją `-O2`. Nie musisz zamieszczać kodu służącego do porównania, ale zamieść w komentarzu jakieś wyniki i wnioski.

Zadanie 3 (4 pkt). Zaimplementuj w monadzie ST strukturę Union-Find. Następnie zdefiniuj typ `Graph` reprezentujący grafy z krawędziami z wagami i zaimplementuj algorytm Prima do znajdowania minimalnego drzewa rozpinającego (algorytm zapewne pamiętają Państwo z AiSD: przechowuj w strukturze Union-Find zbiory już połączonych wierzchołków, a potem kolejno przeglądaj krawędzie od najlżejszej i każdą dorzucaj do drzewa jeśli łączy dwa do tej pory rozłączne zbiory). Niech algorytm zawarty będzie w funkcji `minSpanningTree :: Graph -> Graph`, a więc o czystym typie (mimo manipulacji wskaźnikami w środku).

Zadanie 4 (1 pkt). Wymyśl czysto funkcyjną implementację Union-Finda. Prawdopodobnie nie będzie ona tak wydajna jak wersja z użyciem mutowalnej pamięci (nie wiadomo czy taka w ogóle istnieje), ale nie przejmuj się tym. Porównaj wydajność algorytmu Prima używając obu implementacji.