

# Programowanie funkcyjne 2016

Grupa kzi  
Lista 8  
(13.12.2016)

UWAGA†: Jeśli zadanie jest niedospecyfikowane, to braki w specyfikacji można uzupełnić w dowolny sensowny(!) sposób.

## ZADANIA W HASKELLU

### 1. Zdefiniuj:

- a) typ `OptGraph a` przechowujący czwórki  $n, e, l, i$ , gdzie  $n :: \text{Int}$ ,  $e :: \text{Int} \rightarrow \text{Set Int}$ ,  $l :: \text{Int} \rightarrow a$ ,  $i :: a \rightarrow \text{Int}$ ,
- b) typ `ListGraph a` przechowujący pary  $v, e$ , gdzie  $v :: [\mathbf{a}]$ ,  $e :: a \rightarrow [\mathbf{a}]$ , (czyli prosty typ dla grafów skierowanych, gdzie  $v$  reprezentuje zbiór wierzchołków, a  $e$  listy sąsiadów)
- c) klasę `Graph`, taką że  $G :: * \rightarrow *$  należy do `Graph`, gdy implementuje funkcje
  - `nodes :: Ord a => G a -> Set a` – zwraca zbiór wierzchołków grafu,
  - `neighbors :: Ord a => G a -> a -> Set a` – dla każdego wierzchołka  $v$  zwraca zbiór wierzchołków  $S_v$ , takich że  $(v, v')$  jest krawędzią w grafie wtw gdy  $v'$  należy do  $S_v$ ,
  - opcjonalnie `opt :: Ord a => G a -> OptGraph a` – jeśli `opt` nie jest zaimplementowana, to domyślnie powinna odtwarzać graf zgodny z (\*) za pomocą `nodes` i `neighbors`,
  - opcjonalnie `graph_eq :: (Eq a, Ord a) => G a -> G a -> Bool` – jeśli `graph_eq` nie jest zaimplementowana przez  $G$ , to domyślnie `graph_eq g1 g2` sprawdza, czy zbiory wierzchołków i zbiory sąsiadów każdego wierzchołka w  $g_1$  i  $g_2$  są sobie równe,
- d) instancję `Graph OptGraph` – zgodną z (\*),
- e) instancję `Graph ListGraph` – zgodną z naturalną semantyką. (7 PKT)

(\*) Typ z punktu (a) reprezentuje grafy skierowane, w których poindeksowaliśmy wszystkie wierzchołki liczbami naturalnymi z przedziału  $[0, \dots, n - 1]$ . Funkcja  $i$  zwraca numer wierzchołka. Funkcja  $l$  zwraca wierzchołek o zadanym numerze. Krawędź z wierzchołka  $v$  do wierzchołka  $v'$  istnieje wtw gdy  $i v'$  należy do  $e (i v)$ .

2. Zdefiniuj:

- a) funkcję `print_graph :: (Ord a, Show a, Graph g) => g a -> IO ()`, która wypisuje graf na standardowe wyjście – przy czym lista wierzchołków i każda z list sąsiadów powinna być w innym wierszu,
- b) funkcję `get_graph :: Ord a => (String -> a) -> IO (Graph a)`, która odczytuje graf ze standardowego wejścia.

Przy czym dla każdej funkcji  $f :: \text{String} \rightarrow a$  takiej, że  $f (\text{show } e) == e$  i dla każdego grafu  $g :: \text{Graph } g \Rightarrow g a$ , jeśli `print_graph g` wypisuje tekst *str*, wtedy po wprowadzeniu *str* na wejście wykonanie

```
do { g' <- get_graph f; return (graph_eq g g') }
```

powinno zwracać `True`. (Tzn. `get_graph` powinien rozkodowywać grafy zakodowane przez `print_graph`.) (7 PKT)

3. Zdefiniuj funkcję `exists_path :: (Ord a, Graph g) => g a -> a -> a -> Bool`, która zaaplikowana  $g v v'$  sprawdza, czy istnieje ścieżka z  $v$  do  $v'$  w  $g$ . Funkcja musi działać w czasie liniowy. (14 PKT)

Uwaga: To zadanie będzie trudne dla kogoś, kto nie pracował wcześniej z monadami. (Mi zajęło parę godzin.) Niestety, jeśli chcemy nauczyć się korzystania z monad, to chyba nie ma innej opcji, jak przez takie zadanie przejść.

Wskazówki do Zadania 3, ale jedna też do Zadania 2 (ta o odroczonej obliczeniach):

- Zrzutować graf wejściowy na `OptGraph` i pracować tylko na indeksach wierzchołków.
- Budować `exists_path` inkrementacyjnie. Np. najpierw napisać funkcję, która tworzy tablicę i zwraca listę pustą. Sprawdzić, czy się otypuje. Dodać nadpisanie jednej pozycji w tablicy. Sprawdzić, czy się otypuje... itd.
- Skorzystać z modyfikowalnych tablic `ST(U)Array` i monady `ST` (większość dokumentacji do tablic jest w opisie klas `IArray` i `MArray`).
  - `ST s a` to monada pozwalająca przeprowadzać obliczenia z modyfikowalnym stanem. Parametr  $s$  to typ stanu. Parametr  $a$  to typ wartości zwracanej przez wykonanie obliczenia wykorzystującego stan.
  - Stan możemy sobie wyobrażać jako blok pamięci. Taki blok tworzony jest przez funkcje `runST`, `runSTArray`, `runSTUArray`. Każdy blok stworzony w ten sposób ma inny typ  $s$ . Dzięki temu po obliczeniu wartości w monadzie `ST`, możemy ją z tej monady wyjąć i nie zepsujemy przezroczystości referencyjnej (to bardzo ważne dla Haskella).

- `ST(U)Array s i e` to typ modyfikowalnych tablic, które są przechowywane w stanie o typie `s`. Parametr `i` to typ indeksów w tablicy (jakby ktoś chciał indeksować tablicę czymś innym niż liczbami naturalnymi). Parametr `e` to typ wartości przechowywanych w tablicy. Dodanie `U` (unboxed) oznacza, że wartości w tablicy nie będą opakowywane. To zmniejsza pamięciożerność i czas dostępu do elementów tablicy, ale takie tablice można stworzyć tylko dla małych wartości (typu `Int`, `Bool`, `Float`, itp.).
- Modyfikowalną tablicę tworzymy za pomocą funkcji `newArray` z klasy `MArray`. Problemem jest tu to, że ta klasa ma wiele instancji, a Haskell nie jest w stanie z argumentów wydedukować, której z nich użyć. Trzeba więc wymusić typ tworzonej tablicy. Np. tak: `newArray (1, 10) 0 :: ST s (STUArray s Int Int)`.
- Jeśli mamy w monadzie jakieś obliczenia, które modyfikują stan, to musimy uważać, żeby nie zostały one odroczone. Np. dla takiej funkcja `too_lazy` będziemy mieli: `too_lazy 1 == 0`, `too_lazy 2 == 0`, `too_lazy 3 == 3`.

```

too_lazy n =
  runST
  (do
    arr <- newArray (1, 3) 0 :: ST s (STUArray s Int Int)
    foldl
      (\state_carrier e -> writeArray arr e e)
      (return ())
      [1,2,3]
    readArray arr n)

```

Czemu tak? Popatrzmy. Ostatnia linijka funkcji wymusza obliczenie stanu zwracanego przez `foldl`'a. Ten stan jest wynikiem obliczenia funkcji związającej na ostatnim elemencie listy `[1,2,3]`. Czyli zostanie wymuszone obliczenie `writeArray arr 3 3`. Ale stan z poprzedniej iteracji `foldl`'a, który jest trzymany w `state_carrier`, nie jest potrzeby do tego obliczenia, więc jego obliczenie pozostanie odroczone. Żeby to naprawić wystarczy przed `writeArray arr e e` dopisać `state_carrier >> .` (Operatory `>>=`, `>>` monady `ST` wymuszają obliczenie stanu.)

- Kiedy używamy funkcji `writeArray`, `readArray` trzeba zagwarantować, że typ ich wyniku to `ST s ...`, gdzie `s` jest taki sam, co `s` w typie tablic, do których je aplikujemy. Inaczej będziemy dostawać błędy w stylu `No instance for (MArray (STUArray s) Int m)`. Żeby tak było wystarczy, żeby wynik każdego obliczenia w monadzie `ST` był zawsze przekazywany do kolejnego obliczenia. Można myśleć, że operator `>>` też przekazuje wynik obliczenia, tylko nie łączy go z żadną zmienną. Dokładniej mówiąc, oba operatory `>>=`, `>>`

w ST przekazują stan, który jest niejawną częścią wyniku każdego obliczenia typu ST  $s \dots$ ).