Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 10

Do zgłoszenia w SKOS-ie do 31 maja 2020

Generowanie struktur i testowanie przy użyciu narzędzia QuickCheck

Zadanie 1 (2 pkt). Rozważ typ drzew:

```
data Tree = Leaf | Single Tree | Double Tree Tree
```

Zdefiniuj funkcję `genTree :: [Spec] -> [Tree]`, gdzie `Spec` to typ danych określający warunek generowania drzewa:

```
data Spec = MinNodes Int  -- co najmniej n konstruktorow
          | MaxNodes Int  -- co najwyzej n konstruktorow
          | MinDepth Int  -- glebokosc co najmniej n
          | MaxDepth Int  -- glebokosc co najwyzej n
          | MinPaths Int  -- co najmniej n sciezek
          | MaxPaths Int  -- co najwyzej n sciezek
```

Funkcja `genTree` przyjmuje jako swój argument listę takich specyfikacji i generuje (być może nieskończoną) listę **wszystkich** drzew, które spełniają koniunkcję tych warunków. Np.

```
> genTree [MinPaths 2, MaxPaths 2, MaxNodes 4]
[ Double Leaf Leaf,
  Double (Single Leaf) Leaf,
  Double Leaf (Single Leaf),
  Single (Double Leaf Leaf) ]

> genTree [MaxDepth 0]
[ Leaf ]

> genTree [MinDepth 2]
[ Single (Single Leaf),
  Single (Double Leaf Leaf),
  Double (Single Leaf) Leaf,
  Double (Double Leaf Leaf) Leaf,
  Double Leaf (Single Leaf), ...
```

Trudność zadania polega na tym, że powinny być generowane wszystkie drzewa, to znaczy, że każde spełniające warunki podane w argumencie musi w którymś momencie wystąpić na liście wynikowej. Jeśli drzew spełniających warunek jest skończenie wiele, lista wynikowa powinna być skończona.

Uwaga: W tym zadaniu proszę nie korzystać z dobrodziejstw modułów `Data.Data` i pakietów typu SmallCheck – chodzi o to, żeby zaimplementować cały mechanizm generowania samemu.

Zadanie 2 (2 pkt). Użyj generatorów z modułu `Generic.Random` do generowania losowych grafów. Użyj QuickChecka do sprawdzenia, że dwie implementacje funkcji znajdującej minimalne drzewo spinające z Zadań 3. i 4. z poprzedniej listy dają taki sam rezultat. Odpowiednio dobierz parametry generowania grafów tak, by generowane grafy nie były trywialne.

Uwaga: Jeśli nie posiadasz rozwiązań zadań 3. i 4. z listy nr 9, przykładowe rozwiązania pokażą się na SKOS-ie.

Zadanie 3 (2 pkt). Zdefiniuj typ danych kolejki FIFO w najbardziej standardowy sposób, przy użyciu pary list:

```
data Queue a = Queue { front :: [a], rear  :: [a] }

empty :: Queue a
pushBack :: a -> Queue a -> Queue a
popFront :: Queue a -> Queue a
peek :: Queue a -> a
isEmpty :: Queue a -> Bool
```

Dodatkowo (żeby zawsze móc podejrzeć pierwszy element bez odwracania listy `rear`) zachowujemy niezmiennik: jeśli lista `rear` jest niepusta, to lista `front` też jest niepusta.

Przetestuj swoją implementację przy użyciu QuickChecka. Przykładowo, możesz zdefiniować mniej wydajną wersję kolejki przy użyciu zwykłej listy i porównać ich działanie na ciągach poleceń `Push`, `Pop` itd. Możesz też wymyślić własne podejście do testowania kolejki. Pamiętaj przetestować, czy niezmiennik jest zachowany.

Zadanie 4 (2 pkt). Monada to konstruktor typu `m :: * -> *` razem z funkcjami

- `return :: a -> m a`
- `(>>=) :: m a -> (a -> m b) -> m b`

spełniającymi pewne prawa. Alternatywnie, powiemy, że monada to `Functor` razem z funkcjami

- `return :: a -> m a`
- `join :: m (m a) -> m a`

które spełniają prawa

- `join . fmap return == id`
- `join . return == id`
- `join . fmap join == join . join`

Wówczas możemy zdefiniować:

- `join m = m >>= id`
- `fmap f m = m >>= (return . f)`

I w drugą stronę:

- `(m >>= f) = join (fmap f m)`

W pliku `Monads.hs` znajduje się kilka definicji kandydujących do bycia monadami. Użyj narzędzia QuickCheck żeby znaleźć kontrprzykłady dla definicji, które nie spełniają powyższych praw. Oznacz komentarzem, które definicje udało Ci się wyeliminować.

Zadanie to możesz rozwiązać edytując plik `Monads.hs` (wówczas rozwiązanie całej listy dołązz jako archiwum zawierające główny plik z rozwiązaniem zadań z listy i `Monads.hs`). Pamiętaj o tym, że prawdopodobnie będziesz musiał poprosić kompilator o zainstalowanie tych typów w QuickCheckowych klasach (np. `Arbitrary`).

Wykorzystanie kwantyfikacji uniwersalnej i egzystencjalnej

Jednym z przydatnych typów danych jest `Yoneda` i `CoYoneda` (`https://en.wikipedia.org/wiki/Nobuo_Yoneda`). Wartości typu `CoYoneda` przechowują konstruktor typu zaaplikowany do typu kwantyfikowanego **egzystencjalnie** (a więc niewidocznego na zewnątrz) razem z funkcją, która potrafi z tego typu egzystencjalnego przejść do znanego typu, będącego argumentem konstruktora typu `CoYoneda`:

```
data CoYoneda f a = forall b. CoYoneda (b -> a) (f b)
```

Dla dowolnego `f :: * -> *`, wartości typu `f a` można zapakować do CoYonedy ustalając typ egzystencjalny na `a`, a funkcję „tłumaczącą" na identyczność:

```
toCoYoneda :: f a -> CoYoneda f a
toCoYoneda x = CoYoneda id x
```

W drugą stronę, by wyciągnąć z CoYonedy wartość `f a` (o ile `f` jest funktorem) wystarczy zaaplikować funkcję „tłumaczącą" w środku `f`:

```
fromCoYoneda :: (Functor f) => CoYoneda f a -> f a
fromCoYoneda (CoYoneda f x) = fmap f x
```

Zadanie 5 (1 pkt). Uczyń typ `CoYoneda f` funktorem. Zrób to ręcznie, nie korzystając z rozszerzeń w stylu `DeriveFunctor` i `StandaloneDeriving`. Spróbuj udowodnić (np. zapisując odpowiednie równościowe wnioskowanie w komentarzu), że dla funktora `f`, typ `f a` jest izomorficnzy z `CoYoneda f a`, a izomorfizm dany jest przez dwie powyższe funkcje. To znaczy zachodzi:

- `fromCoYoneda . toCoYoneda = id`
- `toCoYoneda . fromCoYoneda = id`

Uwaga: w zadaniu jest napisane „spróbuj" bo jedna z tych równości łatwo nie pójdzie. Jeśli nie pójdzie, nie przejmuj się tym i udowodnij tylko tę drugą.

Zadanie 6 (2 pkt). Po co nam cały ten `CoYoneda`? Pierwszym zastosowaniem jest to, żeby opóźnić robienie `fmap`. Wywołanie `fmap` zwykle przebudowuje całą strukturę danych, np kopiuje całe drzewo/listę, by zmienić wartości w liściach/elementach. Jeśli robimy kilka `fmap`-ów z rzędu, musimy kilka razy przebudować strukturę, co spowalnia wykonanie i tworzymy nieużytki. Kompilator Haskella na szczęście wie, że `fmap f . fmap g = fmap (f . g)` i jak tylko może, stosuje tę optymalizację. Ale czasem statycznie nie wiadomo, że będzie kilka wywołań funkcji `fmap` z rzędu i wtedy trzeba ręcznie zrobić tę optymalizację – do służy `CoYoneda`. Zamiast wykonywać `fmap`-y, kumulujemy je w pierwszym argumencie konstruktora `CoYoneda` i dopiero gdy potrzebujemy poznać strukturę, aplikujemy jednego `fmap`-a wykonującego kilka złożonych ze sobą funkcji naraz.

W tym zadaniu rozważ typ reprezentujący składnię abstrakcyjną prostego języka programowania ze zmiennymi:

```
data Expr a = Add  (Expr a) (Expr a)
            | Mult (Expr a) (Expr a)
            | Var  a
 deriving (Show, Functor)
```

Zadaniem jest napisać proste narzędzie do refaktoryzacji tego języka. Zdefiniuj funkcję `refactor :: Expr String -> IO ()`. Po uruchomieniu, nasze narzędzie przechowuje w pamięci program będący argumentem i w pętli wczytuje z wejścia standardowego jedną z poniższych instrukcji:

- `/ident1/ident2` – zmień w programie nazwę zmiennej `ident1` na `ident2`.
- `camel` – zmodyfikuj w programie wszystkie nazwy zmiennych tak, by były w camelCase[1].
- `snake` – zmodyfikuj w programie wszystkie nazwy zmiennych tak, by były w snake_case[2].
- `print` – wypisuje program na wyjście standardowe

Przykładowe użycie:

[1] Nie wnikajmy w przypadkologię, jaka jest dokładna definicja. Proszę zrobić tak, żeby działało w tych najoczywistszych przypadkach

[2] Jak wyżej

```
> run $ Add (Var "hello") (Mult (Var "good_morning") (Var "goodnight"))
/goodnight/good_night
camel
/hello/helloThere
snake
print
Add (Var "hello_there") (Mult (Var "good_morning") (Var "good_night"))
camel
print
Add (Var "helloThere") (Mult (Var "goodMorning") (Var "goodNight"))
```

By uniknąć budowania całego programu od nowa przy okazji każdej instrukcji refaktoryzatora, do wewnętrznej reprezentacji programów nie używaj `Expr String` tylko `CoYoneda Expr String`. Program buduj tylko przy okazji instrukcji `print`.

Zadanie 7 (1 pkt). Kolejnym zastosowaniem coYonedy jest to, że umożliwia ona potraktowanie dowolnego konstruktora typu jak funktora, nawet jeśli nie jest on funktorem. Przykładowo, listy `[x]` są funkcotrem w bardzo oczywisty sposób. Listy różnicowe `[x] -> [x]` optymalizują listy umożliwiając szybką konkaetnację, ale w oczywisty sposób funktorem nie chcą być. Jako przykład rozważmy strukturę generycznego drzewa z etykietami w liściach, którego struktura wewnętrznego węzła określona jest przez funktor (jest to tzw. *wolna monada*):

```
data Free f a = Node (f (Free f a))
              | Var a
 deriving (Functor)
```

Np. zwykłe drzewo binarne można uzyskać tak:

```
data Pair x = Pair x x deriving (Functor, Foldable)
type BinTree = Free Pair
```

Definiujemy funkcję, która sumuje wartości w liściach:

```
sumFree :: (Functor f, Foldable f) => Free f Int -> Int
sumFree (Var n)  = n
sumFree (Node s) = foldr' (+) 0 $ fmap sumFree s
```

Np.

```
> sumFree $ Node $ Pair
                     (Var 4)
                     (Node $ Pair (Var 2) (Var 7))
13
```

W przypadku listy:

```
> sumFree $ Node
    [ Var 4
    , Node [Var 2, Var 6, Var 1]
    , Node [Var 10]
    , Var 1
    ]
24
```

Gorzej z listą różnicową:

```
newtype DList a = DList { unDList :: [a] -> [a] }

instance Foldable DList where
```

```
  foldr f x xs = foldr f x $ unDList xs []

{- to nie zadziała, bo DList nie jest funktorem
sumFree $ Node $ DList $ \xs ->
  [ Var 4
  , Node $ DList $ \xs -> [Var 2, Var 6, Var 1] ++ xs
  , Node $ DList $ \xs -> [Var 10] ++ xs
  , Var 1
  ] ++ xs
-}
```

Można więc spróbować tak:

```
cydList xs = toCoYoneda $ DList $ (xs ++)

test = sumFree $ Node $ cydList
         [ Var 4
         , Node $ cydList [Var 2, Var 6, Var 1]
         , Node $ cydList [Var 10]
         , Var 1
         ]
```

Wystarczy teraz zdefiniować:

```
instance (Foldable f) => (Foldable (CoYoneda f)) where ...
```

I to właśnie należy zrobić w tym zadaniu.