Kurs języka Haskell 2024/25

LISTA NR 3 (TERMIN: 28.10.2024, godz 20:00)

Uwaga: Wszystkie rozwiązania należy umieścić w jednym module o nazwie `Lista3` zachowując sygnatury zgodne z szablonem rozwiązań zamieszczonym w SKOS-ie.

Zadanie 1. Rozważmy drzewa binarne z etykietami w liściach:

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

Inspirując się funkcją repmin z wykładu, zdefiniuj funkcję

```haskell
flipVals :: Tree a -> Tree a
```

która odwraca kolejność etykiet w liściach drzewa, nie zmieniając kształtu drzewa. Na przykład:

```haskell
ghci> let t = Node (Leaf 't') (Node (Leaf 'a') (Leaf 'k'))
ghci> flipVals t
Node (Leaf 'k') (Node (Leaf 'a') (Leaf 't'))
```

Funkcji flipVals wolno przejść przez drzewo tylko raz.

Wskazówka: W przypadku dla liścia może przydać się leniwe dopasowanie wzorca, o którym mówiliśmy na pracowni.

Zadanie 2. Rozważmy typ list różnicowych:

```haskell
newtype DiffList a = DiffList { unDiffList :: [a] -> [a] }
```

(**newtype** działa mniej więcej jak **data**, ale pozwala na jeden konstruktor z jednym argumentem – więcej o **newtype**'ach będzie mówione na wykładzie.)

Lista różnicowa reprezentuje listę w postaci funkcji, której argument staje się ogonem reprezentowanej listy. Przykładowo, lista

```haskell
[1, 2, 3]
```

reprezentowana jest jako:

```haskell
DiffList (\ys -> 1 : 2 : 3 : ys)
```

Celem istnienia list różnicowych jest pozbycie się liniowej złożoności konkatenacji.

W tym zadaniu:

- Zdefiniuj funkcje

```haskell
fromDiffList  :: DiffList a -> [a]
toDiffList    :: [a] -> DiffList a
diffSingleton :: a -> DiffList a
```

  które konwertują do i ze zwykłych list i funkcję, która tworzy jednoelementową listę różnicową. Postaraj się zdefiniować je *bezpunktowo*, to znaczy bez nazywania argumentów, a jedynie jako złożenia innych funkcji.
- Zainstaluj listy różnicowe w klasach Semigroup i Monoid.
- Rozważ drzewa zdefiniowane jak w poprzednim zadaniu. Zainstaluj konstruktor typu Tree w klasie Foldable, która pozwala na „spłaszczenie" drzewa do dowolnego monoidu.
- Porównaj czas wykonania, jeśli do spłaszczania drzewa użyjemy listy i listy różnicowej. To znaczy: porównaj

```haskell
foldMap (\x -> [x]) t
```

  z

```haskell
fromDiffList (foldMap diffSingleton t)
```

- Porównaj czas wykonania z ogonowym rozwiązaniem z akumulatorem.

Informację o prostym sposobie mierzenia czasu wykonania można znaleźć we skazówce do Zadania 5. z Listy 2. Pamiętaj, żeby przypadkiem nie zmierzyć czasu tworzenia samego drzewa w pamięci (szczególnie, że może ono zostać tworzone leniwie w momencie pierwszego pomiaru, zupełnie zakłamując wynik).

Ciekawostka: Kompilator sam umie zainstalować typ Tree w klasie Foldable, a także DiffList w klasach Semigroup i Monoid przy użyciu polecenia **deriving**. Oczywiście w tym zadaniu należy to zrobić samodzielnie.

Wskazówka: W bezpunktowych definicjach funkcji fromDiffList itp. przydać się może infiksowy operator

```haskell
($) :: (a -> b) -> a -> b
```

który aplikuje funkcję do argumentu.

Zadanie 3. Rozważmy typ danych reprezentujący binarne drzewa stochastyczne:

```
data ProbTree a
  = PTLeaf a
  | PTNode Double (ProbTree a) (ProbTree a)
```

Intuicyjnie, wartość `PTNode p t1 t2` należy rozumieć jako wybór pomiędzy `t1` a `t2`, przy czym `t1` wybierany jest z prawdopodobieństwem `p`, a `t2` z prawdopodobieństwem $1 - p$. W liściach znajdują się końcowe wartości prcesu opisanego przez drzewo.

Rozważmy też podobny typ, w kótrym prawdopodobieństwo obu gałęzi jest zawsze równe 0.5:

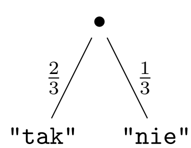
```
data CoinTree a
  = CTLeaf a
  | CTNode (CoinTree a) (CoinTree a)
```

W tym zadaniu:

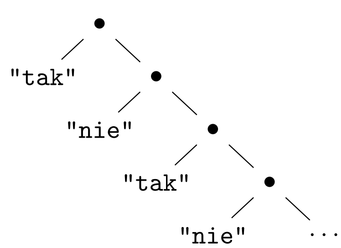
- Zdefiniuj funkcję

  ```
  toCoinTree :: ProbTree a → CoinTree a
  ```

  Która przetwarza dowolne drzewo stochastyczne na drzewo, w którym każdy wybór dokonywany jest przez rzut monetą. Drzewo wynikowe może oczywiście być nieskończozne, np. proces

  może stać się procesem

  Dopilnuj, by wartości w dzieciach tak rozwiniętego procesu były dzielone a nie duplikowane!
- Zdafiniuj funkcję

  ```
  coinRun :: [Bool] → CoinTree a → a
  ```

  która uruchamia proces przedstawiony za pomocą wartości typu `CoinTree` przy użyciu nieskończonego strumienia „losowych” bitów (ustalamy, że **True** „idzie” w lewo, a **False** – w prawo).

Uwaga: Możemy uruchomić taki proces przy pomocy generatora liczb pseudolosych zainicjowanego przy pomocy `/dev/urandom` używając monady **IO** (bez **IO** nie mamy szans dobrać się ani do czasu systemowego ani do randomów):

```
import qualified Data.List as List
import qualified System.Random as Random

-- ...

randomCoinRun :: CoinTree a → IO a
randomCoinRun t = do
  gen ← Random.initStdGen
  let bitStream =
    List.unfoldr (Just . Random.uniform) gen
  return (coinRun bitStream t)
```

gdzie:

- Żeby powyższe działało, może być konieczne zainstalowanie biblioteki `random`:

  ```
  cabal install --lib random
  ```

- O **do** i **return** będziemy jeszcze mówić,
- Proszę zwrócić uwagę, że nigdzie nie mówimy explicite, jakiego typu mają być rzeczy generowane przez **Random**.uniform. Są one wybierane z typów, które zainstalowane są w klasie **System**.**Random**.Uniform. Kompilator wybiera spośród tych typów **Bool**, ponieważ inferencja typów mówi mu, że bitStream ma mieć typ [**Bool**], ze względu na użycie go jako argumentu funkcji coinRun.

Bardzo ważna uwaga: Proszę zwrócić uwagę, że w językach z efektami pewnie połączylibyśmy generowanie kolejnych pseudolosowych bitów z przechodzeniem drzewa od razu w funkcji coinRun. Jednak w Haskellu nie możemy tego zrobić, nie umieszczając coinRun w monadzie **IO**. Ponieważ zwykle chcemy, żeby nieczysty fragment programu był jak najmniejszy, a jak najwięcej funkcji było bez efektów, musimy skorzystać z pośrednika z postaci strumienia bitów. Jest to klasyczne Haskellowe rozwiązanie, gdzie oddzielamy część z efektami (generowanie bitów) od samej logiki funkcji (przechodzenie drzewa).

Dodatkowo, proszę się zastanowić, jak zdefiniowalibyśmy matematycznie semantykę drzewa losowego procesu (np. pisząc podręcznik do matematyki dla liceum). Czy ta definicja nie byłaby przypadkiem funkcją coinRun (modulo składnia)?

Zadanie 4. Typ danych

```
data Frac = Frac Integer  Integer
```

może być użyty do reprezentowania ułamków. Zainstaluj go w klasie **Num**. Spraw, by ułamki po wykonaniu operacji, zawsze były przedstawione w postaci znormalizowanej.

Zadanie 5. Typ danych

```
data CReal = CReal { unCReal :: [Frac] }
```

(gdzie Frac to typ z poprzedniego zadania) może być użyty do reprezentowania liczb rzeczywistych jako ciągu kolejnych przybliżeń. Zainstaluj go w klasie **Num**.

Zdefiniuj wartość

```
realPi  ::  CReal
```

reprezentującą liczbę π.