Kurs języka Haskell 2024/25

LISTA NR 8 (TERMIN: 7.01.2025, godz 5:00)

Uwaga: Wszystkie rozwiązania należy umieścić w jednym module o nazwie `Lista8` zachowując sygnatury zgodne z szablonem rozwiązań zamieszczonym w SKOS-ie.

Zadanie 1. Często wygodniej pracować z konkretnym typem danych niż instancją bardziej generycznej struktury. Przykładowo, często czytelniej jest pracować z typem

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

niż

```
data Pair x = Pair x x
type Tree = Term Pair
```

Z drugiej strony, być może mamy już zestaw użytecznych funkcji na typie `Term`, które chcielibyśmy wykorzystać w naszym kodzie. Złoty środek to zdefiniowanie klasy typów, której instancjami są zarówno `Term`, jak i jego bardziej konkretni kuzyni w rodzaju `Tree`, a następnie zdefiniowanie przydatnych funkcji dla instacji tej klasy.

Zdefiniuj dwuargumentową klasę `Term t sig`, która zawiera metody

```
var :: x -> t x
op :: sig x -> t x
subst :: t x -> (x -> t y) -> t y
foldTree :: (sig a -> a) -> (x -> a) -> t x -> a
```

Zdefiniuj instancje tej klasy

- **instance** (Term sig) sig dla dowolnego funktora sig
- **instance** Tree Pair

Zdefiniuj funkcję

```
collectVars :: (Foldable s, Term t s, Monoid m)
            => t m -> m
```

która zwija wszystkie wartości w liściach przy użyciu monoidu.

Następnie wykorzystaj rozszerzenie `UndecidableInstances`, żeby zainstalować wszystkie instancje klasy `Term` w klasie **Monad**. (Ta operacja jest dość kontrowersyjna, zobacz https://stackoverflow.com/questions/3079537/orphaned-instances-in-haskell).

Wskazówka: Być może chcez dodać do tej klasy zależność funkcyjną (*functional dependency*).

Zadanie 2. Wyrażenia często definiuje się przez składnię i semantykę. Składnia zwykle zadana jest przez algebraiczny typ danych, np.

```
data Expr = Val Int
          | Add Expr Expr

ex1 :: Expr
ex1 = Add (Add (Val 3) (Val 9)) (Val 10)
```

Semantykę oraz inne funkcje operujące na składni można zdefiniować przez zwijanie drzewa składni, np.

```
eval :: Expr -> Int
eval (Val n)   = n
eval (Add l r) = eval l + eval r

pprint :: Expr -> String
pprint (Val n)   = show n
pprint (Add l r) = "(" ++ pprint l ++ " + "
                       ++ pprint r ++ ")"
```

Innym sposobem jest tzw. reprezentacja *final tagless*. Zakładamy w niej, że termy reprezentują wartości w pewnym semantycznym type `d` (dziedzinie), a term budujemy z semantycznych reprezentacji operacji, czyli używając metod z następującej klasy:

```
class ExprFT d where
  val :: Int -> d
  add :: d -> d -> d

ex2 :: ExprFT d => d
ex2 = add (add (val 3) (val 9)) (val 10)
```

Konkretną semantykę zadajemy przez interpretację tak zbudowanego termu w konkretnej dziedzinie, czyli, nazywając rzecz technicznie, ukonkretnienie jego wartości w jakieś instancji klasy `ExprFT`.

Zdefiniuj funkcję

```
evalFT :: ExprFT d => d -> d
```

która oblicza wartość termu w danej dziedzinie, a następnie zainstaluj w klasie `ExprFT` typy **Int** oraz **String**, odpowiadające funkcjom `eval` i `pprint`, tak, że:

```
ghci> evalFT @Int ex2
22
ghci> evalFT @String ex2
"((3 + 9) + 10)"
```

Zdefiniuj także konwersje między reprezentacjami:

```
ghci> evalFT @Expr ex2
Add (Add (Val 3) (Val 9)) (Val 10)
ghci> evalFT @Int (toFT ex1)
22
```

Uwagi:

- Uwaga na *monomorphism restriction* podczas definiowania wyrażeń *final tagless*: proszę pamiętać o sygnaturach typowych.
- Proszę zwrócić uwagę na typ evalFT: oznacza on, że dane wyrażenie można zinterpretować w *dowolnym* typie semantycznym (dowolnej instancji klasy ExprFT), a nie, że jest on interpretowany w *pewnym* typie należącym do tej klasy. Proszę przypomnieć sobie dyskusję z wykładu o różnicach między klasami typów a klasami w programowaniu obiektowym.

Zadanie 3. Rozszerz powyższy interpreter *final tagless* o wyrażenia boolowskie. Niech język zawiera przynajmniej operacje: stała, dodawanie, porównanie, negacja, if.

Wskazówka: Zdefiniuj klasę dwuargumentową ExprFT a b (mogą przydać się zależności funkcyjne), a następnie funkcje ewaluujące wyrażenia:

```
evalA :: ExprFT a b ⇒ a → a
evalB :: ExprFT a b ⇒ b → b
```

Zaimplementuj ewaluację i pretty-printing.

Zadanie 4. Rozbuduj powyższe zadanie o zmienne w wyrażeniach i polecenia języka WHILE znanego z poprzednich list zadań.

Zadanie 5. Rozważ następującą klasę typów:

```
class Sum a where
  sum :: Integer → a
```

Zainstaluj w niej typ **Integer** oraz typ **Integer** → b dla odpowiedniego b, żeby otrzymać funkcję sumującą dowolną liczbę argumentów, np.

```
ghci> sum 1 :: Integer
1
ghci> sum 1 2 3 4 :: Integer
10
```

Zadanie 6. Rozważ klasę typów

```
class Flatten a b where
  flatten :: a → [b]
```

Użyj jej do zdefiniowania funkcji spłaszczającej listy i pary zagnieżdżone w sobie dowolną liczbę razy, np.

```
ghci> flatten "Ala ma kota" :: String
"Ala ma kota"
ghci> flatten ["Ala", "ma", "kota"] :: String
"Alamakota"
ghci> flatten [["Ala", "ma", "kota"], ["i", "psa"]]
    :: String
"Alamakotaipsa"
ghci> flatten [("Ala", "ma")] :: String
"Alama"
ghci> flatten (("Ala", "ma"), "kota") :: String
"Alamakota"
ghci> flatten (("Ala", ["ma"]), "kota") :: String
"Alamakota"
```

Zadanie 7. Zdefiniuj typ danych (GADT), który zachowuje się jak lista, ale dodatkowo każda wartość oznaczona jest kolorem białym lub czerwonym. Wyraź w typach następujący niezmiennik: *w każdym sufiksie listy, liczba wartości czerwonych jest niewiększa niż liczba wartości białych.*

Przykładowo, wyrażanie w stylu

```
White 'a' . Red 'b' . White 'c' $ Nil
```

powinno się otypować, ale próba kompilacji wyrażenia

```
White 'a' . Red 'b' . Red 'c' . White 'd' $ Nil
```

powinna zakończyć się błędem typów.

Zadanie 8. Jak w poprzednim zadaniu, ale niezmiennik to: *w każdym prefiksie listy, liczba wartości czerwonych jest niewiększa niż liczba wartości białych.*

Wskazówka: W tym wypadku dobrze owinąć zdefiniowany GADT w newtype'a, żeby powiedzieć kompilatorowi, od czego zacząć liczyć.