Metody Programowania: Lista 9

Due on Monday, May 04, 2015

TWI 17.00-19.00

Bartosz Bednarczyk

Spis treści

Metody programowania 2015

Lista zadań nr 9

Na zajęcia 4–6 i 13 maja 2015

Niech

```
reverse, rev :: [a] → [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
rev = aux [] where
   aux ys [] = ys
   aux ys (x:xs) = aux (x:ys) xs
```

Zadanie 1 (1 pkt). Pokaż, że reverse = rev.

Zadanie 2 (1 pkt). Pokaż, że dla dowolnej *skończonej* listy *xs* zachodzi równość:

$$\texttt{reverse}\,(\texttt{reverse}\,xs) \quad = \quad xs$$

Wskazówka: udowodnij wpierw odpowiedni lemat o funkcji ++. Wskaż miejsce w którym dowód twierdzenia bez założenia o skończoności listy *xs* zacina się.

Zadanie 3 (1 pkt). Udowodnij, że dla dowolnej *częściowej* listy *xs* mamy:

$$xs \mathbin{{+}{+}} ys \quad = \quad xs$$

gdzie *ys* jest dowolną listą. Pokaż, gdzie dowód zacina się dla list skończonych.

Zadanie 4 (1 pkt). Na czym polega problem w poniższej definicji funkcji Fibonacciego:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fin n = fib (n-1) + fib (n-2)
```

Zaprogramuj funkcję która wywołana z parametrem n wykonuje $O(\mathrm{n})$ iteracji w celu obliczenia wartości fib n.

Zadanie 5 (1 pkt). Zaprogramuj w Haskellu funkcję

```
roots :: (Double, Double, Double) -> [Double]
```

wyznaczającą listę miejsc zerowych trójmianu kwadratowego o podanych współczynnikach. *Wskazówka:* typ Double należy do klasy Ord, zdefiniowano więc dla niego metodę

```
compare :: Double -> Double -> Ordering
```

gdzie

```
data Ordering = LT | EQ | GT
   deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

W preludium standardowym zdefiniowano też funkcję

```
sqrt :: (Floating a) => a -> a
```

a typ Double należy do klasy Floating. Niech

```
data Roots = No
           | One Double
           | Two (Double, Double)
           deriving Show
roots :: (Double, Double, Double) -> Roots
```

Zaprogramuj tę funkcję. Czy jest lepsza niż poprzednia? Podaj argumenty za i przeciw. Skomentuj następnie funkcje o sygnaturach:

```
roots :: Double -> Double -> Double -> [Double]
roots :: [Double] -> [Double]
```

Zadanie 6 (1 pkt). Nie korzystając z faktu, że typ Integer należy do klasy Show zaprogramuj funkcję

```
integerToString :: Integer -> String
```

Wskazówka: wykorzystaj funkcję

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

z modułu List daną wzorem

```
unfoldr f b =
   case f b of
      Nothing    -> []
      Just (a,b) -> a : unfoldr f b
```

gdzie

```
data Maybe a = Nothing
             | Just a
             deriving (Eq, Ord, Read, Show)
```

Moduł Char udostępnia funkcję

```
intToDigit :: Int -> Char
```

preludium standardowe oferuje następującą metodę klasy Enum:

```
fromEnum :: (Enum a) => a -> Int
```

a typ Integer należy do klasy Enum.

Zadanie 7 (1 pkt). Zbiory, także nieskończone, można reprezentować w postaci ich funkcji charakterystycznych:

```
newtype FSet a = FSet (a -> Bool)
```

Zdefiniuj następujące operacje:

```
empty :: FSet a
singleton :: Ord a => a -> FSet a
fromList :: Ord a => [a] -> FSet a
union :: Ord a => FSet a -> FSet a -> FSet a
intersection :: Ord a =>
                  FSet a -> FSet a -> FSet a
member :: Ord a => a -> FSet a -> Bool
```

Zadanie 1

```
rev, reverse :: [a] -> [a]

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

rev x  = aux [] x  where
         aux ys [] = ys
         aux ys (x:xs) = aux (x:ys) xs

main = putStrLn $ show( rev [1,2,3]  )
```

Twierdzenie 1. *rev = reverse*

Dowód. Z zasady ekstensjonalności, wiemy że dwie funkcje są równe, jeżeli są równych typów oraz dla każdego argumentu przyjmują taką samą wartość.
Umocnijmy tezę i zapiszmy ją jako

$$(\star)\quad aux\ R\ L = (reverse\ L)\ ++\ R.$$

Łatwo zauważyć, że ($\star$) implikuje twierdzenie 1.
Weźmy dowolną listę R, a dowód poprowadźmy indukcyjnie względem struktury listy L.

- $L = [\ ]$ - widać, że funkcje zwracają to samo.
- $L = \perp$ - do dopracowania.
- Weźmy dowolną listę $L = x : xs$ i załóżmy prawdziwość tezy dla listy xs.

$$reverse(x : xs) ++R = (reverse(xs) ++[x])\ ++\ R \overset{lem1}{=} reverse\ xs\ ++\ (x : R) =$$
$$= aux\ R\ L = aux\ R\ (x : xs) \overset{def}{=} aux\ (x : R)\ xs \overset{zal}{=} reverse\ xs\ ++\ (x : R),$$

co dowodzi ($\star$).

Podstawiając za R w ($\star$) listę pustą dostajemy tezę zadania.

□

Zadanie 2

Lemat 1. *Dla dowolnych skończonych list L i K prawdą jest, że $reverse(K \ ++\ L) = reverse(L) \ ++\ reverse(K)$.*

Dowód. Weźmy dowolną skończonąlistę K. Przeprowadzimy dowód indukcyjny względem długości listy L.

- Dla pustej listy L twierdzenie jest poprawne.
- Weźmy dowolne n naturalne i załóżmy, że dla dowolnej listy L o rozmiarze n lemat jest prawdziwy. Udowodnijmy jego prawdziwości dla listy o $n+1$ elementach. Weźmy $L = x : xs$, gdzie xs ma n elementów.

$$reverse(K \ ++\ L) = reverse(\ K \ ++\ [x] \ ++\ xs) = reverse((K \ ++\ [x]) \ ++\ xs) \overset{zal}{=}$$

$$\overset{zal}{=} reverse\ xs \ ++\ reverse(K \ ++\ [x]) \overset{zal}{=} reverse\ xs \ ++\ reverse\ [x] \ ++\ reverse\ K =$$

$$\overset{zal}{=} reverse(L) + reverse(K)$$

□

Twierdzenie 2.

$$reverse(reverse\ xs) = xs$$

Dowód. Niech

$$X = \left\{ n \in \mathbb{N} \ \middle|\ \forall_{L::[a],\ |L|=n}\ reverse(reverse(L)) = L \right\}$$

.

- $n = 0$. Wtedy mamy $reverse(\ reverse[\ ]\ ) = reverse[\ ] = [\ ]$. Stąd $0 \in X$.
- Weźmy dowolne $n \in \mathbb{N}$. Załóżmy, że $n \in X$ i pokażmy, że $n+1 \in X$. Weźmy dowolną listę L o długości $n+1$. Ponieważ jej długość jest niezerowa, to $L = x : xs$, dla pewnych x i xs. Zauważmy, że $|xs| = n$. Zatem

$$reverse(\ reverse\ L\ ) = reverse(reverse\ x : xs\ ) = reverse(\ reverse\ xs \ ++\ [\ x\ ]) \overset{lem}{=}$$

$$\overset{lem}{=} reverse([\ x\ ]) \ ++\ reverse(reverse\ xs) = x \ ++\ reverse(reverse\ xs) \overset{zal}{=}$$

$$\overset{zal}{=} x \ ++\ xs = L$$

Stąd $n+1 \in X$, co na mocy indukcji daje nam, że $X = \mathbb{N}$ - teza jest prawdziwa dla dowolnych skończonych list.

□

Zadanie 3

Zadanie 3 continued on next page. . .

Zadanie 4

```
-- zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
-- Zipuje dwie listy laczac je podanym operatorem
-- Input: zipWith (+) [1,2,3] [3,2,1]
-- Out: [4,4,4]


fib :: Int -> Int

fib n = fibbs !! (n+1) where
        fibbs = 1:1:zipWith (+) fibbs (tail fibbs)

-- Przyklad dzialania :

main = putStrLn $ show( map fib [0,1,2,3,4,5,6,7,8,9,10] )
```

Zadanie 5

```
roots :: (Double, Double, Double) -> [Double]
roots (a, b, c) =
                if a == 0 then
                        if b == 0 then []
                        else [-c/b]

                else
                        case compare delta 0 of
                        EQ -> [-b/(2*a)]
                        LT -> []
                        GT -> [(-b+sqrt(delta))/(2*a), (-b-sqrt(delta))/(2*a)]
                        where delta = b*b-4*a*c



data Roots = No | One Double | Two (Double, Double) deriving Show
roots2 :: (Double, Double, Double) -> Roots
roots2 (a, b, c) =
    if a == 0 then
                        if b == 0 then No
                        else One(-c/b)
        else
        case compare delta 0 of
        EQ -> One (-b/(2*a))
        GT -> Two ((-b - sqrt(delta))/(2*a), (-b + sqrt(delta))/(2*a))
        LT -> No
        where delta = (b*b) - (4 * a * c)

roots3 :: Double -> Double -> Double -> [Double]
roots3 a b c =
                if a == 0 then
                        if b == 0 then []
                        else [-c/b]

                else
                        case compare delta 0 of
                        EQ -> [-b/(2*a)]
                        LT -> []
                        GT -> [(-b+sqrt(delta))/(2*a), (-b-sqrt(delta))/(2*a)]
                        where delta = b*b-4*a*c
```

Zadanie 5 continued on next page...

Zadanie 6

```
import Data.List
import Data.Char

integerToString :: Integer -> String
integerToString 0 = "0"
integerToString n =
(reverse .unfoldr (\n ->
if n == 0 then Nothing
else Just ((intToDigit . fromEnum) (n `mod` 10), n `div` 10)
)) n

main = putStrLn $ show(integerToString 1234567)
```

Zadanie 6 continued on next page...

Zadanie 7

```
newtype FSet a = FSet (a -> Bool)

empty :: FSet a
empty = FSet (\_ -> False)

singleton :: Ord a => a -> FSet a
singleton x = FSet ((==) x)

fromList :: Ord a => [a] -> FSet a
fromList xs = FSet (\x -> x `elem` xs)

union :: Ord a => FSet a -> FSet a -> FSet a
union (FSet a) (FSet b) = FSet (\x -> a x || b x)

intersection :: Ord a => FSet a -> FSet a -> FSet a
intersection (FSet a) (FSet b) = FSet (\x -> a x && b x)

member :: Ord a => a -> FSet a -> Bool
member e (FSet f) = f e
```

Zadanie 7 continued on next page...