Haskell Live

[07] Aufgabenblatt 4 (Bäume) und "What the $(.)\x → x?!"

Bong Min Kim
e0327177@student.tuwien.ac.at

Christoph Spörk
christoph.spoerk@inode.at

Florian Hassanen
florian.hassanen@googlemail.com

Bernhard Urban
lewurm@gmail.com

19. November 2010

What the $(.)\x → x?!

```
  -- ($), (.) und flip sind bereits in Prelude
  -- definiert, wir wollen sie aber selbst definieren
import Prelude hiding (($), (∘), flip)
import Data.Char
import Data.List

foolist :: [Integer]
```

$foolist = [1\,..\,1000]$

$uselessfkt0 :: Integer \to [Integer] \to Integer$
$uselessfkt0\ x\ l = sum\ (filter\ tollespraedikat\ l)$
  where $tollespraedikat\ y = (y\ `mod`\ x) \equiv 0$

Lambda Expressions

-- in hugs: > `(\x -> x + x) 5` = 10
-- soll an $\lambda x \to x + x$ erinnern

$myadd :: Integer \to Integer \to Integer$
$myadd = (\lambda x\ y \to x + y)$
-- in hugs: myadd 2 4 = 6

$uselessfkt1 :: Integer \to [Integer] \to Integer$
$uselessfkt1\ x\ l = sum\ (filter\ (\lambda y \to (y\ `mod`\ x) \equiv 0)\ l)$

-- oder gesamte Funktion als Lambda Expression
$uselessfkt2 :: Integer \to [Integer] \to Integer$
$uselessfkt2 = \lambda x\ l \to sum\ (filter\ (\lambda y \to (y\ `mod`\ x) \equiv 0)\ l)$

`($)`

-- aus der Prelude Definition: Applikationsoperator
$(\$) :: (a \to b) \to a \to b$
$f\ \$\ x = f\ x$

-- schwächste Priorität für *($)*
infixr 0 $

-- ⇒ Klammern sparen! (Lisp hat uns ohnehin schon zu viele gekostet)
$uselessfkt3 :: Integer \to [Integer] \to Integer$

$uselessfkt3\ x\ l = sum\ \$\ filter\ (\lambda y \rightarrow (y\ `mod`\ x) \equiv 0)\ l$

-- besonders bei vielen Funktionsapplikationen praktisch...

$yauf1, yauf2 :: String$

$yauf1 = take\ 4\ (snd\ (splitAt\ 6\ (\texttt{"for"} +\!\!+ \texttt{"teh"} +\!\!+ \texttt{"lulz"} +\!\!+ \texttt{"haha"})))$

$yauf2 = take\ 4\ \$\ snd\ \$\ splitAt\ 6\ \$\ \texttt{"for"} +\!\!+ \texttt{"teh"} +\!\!+ \texttt{"lulz"} +\!\!+ \texttt{"haha"}$

`flip`

-- aus der Prelude Definition

$flip :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

$flip\ fkt\ x\ y = fkt\ y\ x$

$substract :: Int \rightarrow Int \rightarrow Int$

$substract\ x\ y = (flip\ (-))\ x\ y$

`(.)`

-- aus der Prelude Definition: Funktionskomposition

$(\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

$(f \circ g)\ x = f\ (g\ x)$

-- stärkste Priorität für *(.)*

infixr $9\ \circ$

$myToUpper1, myToUpper2, myToUpper3 :: Char \rightarrow Char$

$myToUpper1\ ch = chr\ \$\ ord\ ch - 0\ x20$

$myToUpper2\ ch = (chr \circ (substract\ 0\ x20) \circ ord)\ ch$

$uselessfkt4 :: Integer \rightarrow [Integer] \rightarrow Integer$

$uselessfkt4\ x\ l = (sum \circ filter\ (\lambda y \rightarrow (y\ `mod`\ x) \equiv 0))\ l$

Pointfree

$myToUpper3 = chr \circ (substract\ 0\ x20) \circ ord$

-- on the way to pointfree...

$uselessfkt5 :: Integer \to [Integer] \to Integer$

$uselessfkt5\ x = sum \circ filter\ (\lambda y \to (y\ `mod`\ x) \equiv 0)$

-- pointfree (thx @lambdabot ;-))

$uselessfkt6 :: Integer \to [Integer] \to Integer$

$uselessfkt6 = (sum \circ) \circ filter \circ flip\ flip\ 0 \circ ((\equiv) \circ) \circ flip\ mod$

Expression	Typ
`(sum .)`	`(b -> [a]) -> b -> a`
`filter`	`(a -> Bool) -> [a] -> [a]`
`flip`	`(a -> b -> c) -> (b -> a -> c)`
`flip flip`	`a -> (b -> a -> c) -> (b -> c)`
`flip flip 0`	`(b -> a -> c) -> (b -> c)`
`(.)`	`(a -> b) -> (c -> a) -> c -> b`
`(==)`	`a -> a -> Bool`
`((==) .)`	`(b -> a) -> b -> a -> Bool`
`flip mod`	`a -> a -> a`

⇒ Pointfree ist nicht immer sinnvoll! Für Interessierte: `http://www.haskell.org/haskellwiki/Pointfree`

Hausaufgabe: Wo könnte der Pointfreestyle bei Aufgabe6 sinnvoll sein?

Aufgabenblatt 4

-- my tree definition

data *Tree* = *Leaf Integer* |
Node Integer Tree Tree **deriving** *Show*

```haskell
type Layer = [Integer]
data MyOrd = BottomUp | TopDown
  -- some trees
t1 = (Node 5 (Node 5 (Leaf 4) (Leaf 2)) (Leaf 3))
t2 = (Node 5 (Node 5 (Leaf 4) (Leaf 2)) (Node 3 (Leaf 1) (Leaf 3)))
t3 = (Node 5 (Node 5 (Leaf 4) (Node 2 (Leaf 1) (Leaf 3))) (Node 5 (Node 5 (Leaf 4) (Node 2 (Leaf 1) (Leaf 1))) (Leaf 3)))
t4 = (Node 1 (Node 1 (Node 1 (Leaf 2) (Leaf 3)) (Node 1 (Leaf 2) (Leaf 3))) (Node 1 (Node 1 (Leaf 2) (Leaf 3)) (Node 1 (Leaf 2) (Leaf 3)
  -- a
mergeLayer :: [Layer] -> [Layer] -> [Layer]
mergeLayer [] r = r
mergeLayer l [] = l
mergeLayer (x1 : x1s) (x2 : x2s) = (x1 ++ x2) : (mergeLayer x1s x2s)
writeLayer :: Tree -> MyOrd -> [Layer]
writeLayer (Leaf x) _ = [[x]]
writeLayer (Node x t1 t2) TopDown = [x] : merged
  where merged = mergeLayer (writeLayer t1 TopDown) (writeLayer t2 TopDown)
writeLayer t BottomUp = reverse $ writeLayer t TopDown
  -- b
data STree = Nil |
  SNode Integer STree STree deriving Show
treeToSortedList :: Tree -> [Integer]
treeToSortedList t = sort $ nub $ foldr (++) [] (writeLayer t TopDown)
splitHalf :: [a] -> ([a], a, [a])
splitHalf l = ((take p l), (l !! (p)), (drop (p + 1) l))
  where p = (length l) `div` 2
listToStree :: [Integer] -> STree
listToStree [] = Nil
listToStree l = SNode x (listToStree l1) (listToStree l2)
  where (l1, x, l2) = splitHalf l
```

```
transform :: Tree → STree
transform t = listToStree (treeToSortedList t)
  -- some tree functions
  -- calculate tree depth
depth :: Tree → Integer
depth (Leaf _) = 0
depth (Node _ subt1 subt2) = 1 + (max (depth subt1) (depth subt2))
flatten :: Tree → [Integer]
flatten (Leaf x) = [x]
flatten (Node x subt1 subt2) = (x : ((flatten subt1) ++ (flatten subt2)))
treemap :: (Integer → Integer) → Tree → Tree
treemap f (Leaf x) = Leaf (f x)
treemap f (Node x subt1 subt2) = Node (f x) (treemap f subt1) (treemap f subt2)
  -- tree printer
space x = map (λx → ' ') [1 . . x]
zeroCopy :: Tree → Tree
zeroCopy (Leaf _) = (Leaf 0)
zeroCopy (Node _ subt1 subt2) = (Node 0 (zeroCopy subt1) (zeroCopy subt2))
setRoot :: Integer → Tree → Tree
setRoot r (Node _ subt1 subt2) = (Node r subt1 subt2)
  -- transformers
  -- balance fills up the given binary tree to full binary tree
balance :: Tree → Tree
balance (Leaf x) = Leaf x
balance (Node x s1@(Node y subt1 subt2) (Leaf z)) = (Node x b1 (setRoot z (zeroCopy b1)))
  where b1 = balance s1
balance (Node x (Leaf z) s2@(Node y subt1 subt2)) = (Node x (setRoot z (zeroCopy b2)) b2)
  where b2 = balance s2
balance (Node x subt1 subt2)
```

```
  | bal1d > bal2d = (Node x bal1 (balance_ bal2 bal1))
  | bal1d < bal2d = (Node x (balance_ bal1 bal2) bal2)
  | otherwise = (Node x bal1 bal2)
  where bal1 = balance subt1
        bal2 = balance subt2
        bal1d = depth bal1
        bal2d = depth bal2

  -- the first tree gets the same structure as the second one
balance_ :: Tree → Tree → Tree
balance_ (Leaf x1) (Leaf x2) = (Leaf x1)
balance_ (Node x s1 s2) (Leaf _) = (Node x s1 s2)
balance_ (Leaf x) b1@(Node y s1 s2) = (setRoot x $ zeroCopy b1)
balance_ (Node x1 s1 s2) (Node _ s3 s4) = (Node x1 (balance_ s1 s3) (balance_ s2 s4))

mergeTreeShow [] [] = []
mergeTreeShow (t1 : t1s) (t2 : t2s)
  | (l `mod` 2) ≡ 0 = ((t1 ++ (space 5) ++ t2) : (mergeTreeShow t1s t2s))
  | otherwise = ((t1 ++ (space 4) ++ t2) : (mergeTreeShow t1s t2s))
  where
    str = (t1 ++ (space 5) ++ t2)
    indices = findIndices (≢ ' ') str
    index1 = indices !! 0
    index2 = indices !! 1
    l = index2 − index1

treeshow_ :: Tree → [String]
treeshow_ (Leaf x) = [show x]
treeshow_ (Node x subt1 subt2) = (help (head mt) x) ++ mt
  where t1 = treeshow_ subt1
        t2 = treeshow_ subt2
        mt = mergeTreeShow t1 t2
        help s x
          | ((index2 − index1) > 2) ∧ (ch ≡ '0' ∨ ch ≡ '-') = (help str2 x) ++ [str2]
```

```
    | ((index2 - index1) > 2) ∧ x ≢ 0 = (help str1 x) ++ [str1]
    | otherwise = [(space (index1 + 1)) ++ (show x) ++ (space (l - index2))]
   where
     indices = findIndices (≢ ' ') s
     index1 = indices !! 0
     index2 = indices !! 1
     ch = s !! index1
     l = length s
     str1 = (space (index1 + 1)) ++ ['/'] ++ (space (index2 - index1 - 3)) ++ ['\\'] ++ (space (l - (index2)))
     str2 = (space (index1 + 1)) ++ ['-'] ++ (space (index2 - index1 - 3)) ++ ['-'] ++ (space (l - (index2)))
treeshow :: Tree → IO ()
treeshow t = sequence_ (map putStrLn (map (map pp) (treeshow_ (balance t))))
  where
    pp c
      | c ≡ '0' ∨ c ≡ '-' = ' '
      | otherwise = c
treeOrigshow :: Tree → IO ()
treeOrigshow t = sequence_ (map putStrLn (treeshow_ $ balance t))
```