Haskell Live

[04] Aufgabenblatt 1 & 2

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

29. Oktober 2010

Hinweise

Diese Datei kann als sogenanntes "Literate Haskell Skript" von `hugs` geladen werden, als auch per `lhs2TeX`[1] und LaTeX in ein Dokument umgewandelt werden.

Algebraische Datentypen

Algebraische Datentypen Definitionen werden mit dem Schlüsselwort *data* eingeleitet. Die simplester Form eines algebraischen Datentyps ist die Enumeration der Elemente

```
import Data.Char
data Temp = Cold | Hot deriving Show
data Season = Spring | Summer | Autumn | Winter
```

Cold und *Hot* werden als Konstruktoren des Typs *Temp* bezeichnet

```
weather :: Season → Temp
weather Summer = Hot
weather _ = Cold
```

Wir können auch einen Datentyp definieren, der aus mehreren Komponenten besteht, und er wird auch als Produkttyp bezeichnet

[1] `http://people.cs.uu.nl/andres/lhs2tex`

```
data People = Person Name Age
type Name = String
type Age = Int

showPerson :: People → String
showPerson (Person st n) = st ++ " -- " ++ (show n)
```

newtype definiert auch einen neuen Datentypen, aber dabei ist nur ein Konstruktor erlaubt und dieser Konstruktor muss genau ein Argument besitzen.

```
newtype Student = Student Name

showStudent :: Student → String
showStudent (Student name) = "Student " ++ name
```

Wir können den Argumenten eines Konstruktors die Labels mitgeben

```
data Point = Pt Float Float
pointx          :: Point → Float
pointx (Pt x _) = x
pointy          :: Point → Float
pointy (Pt _ y) = y
  -- alternativ
data Point2 = Pt2 {point2x, point2y :: Float}
```

Mit *rekursiven* algebraischen Datentypen sind auch unendliche Datenstrukturen möglich wie Listen und Baeume etc.

```
data MyList = Nil |
  Elem Integer MyList

testlist :: MyList
testlist = (Elem 1 (Elem 2 (Elem 3 Nil)))

myListSize :: MyList → Int
myListSize Nil = 0
myListSize (Elem x rest) = 1 + (myListSize rest)

myListToList :: MyList → [Integer]
myListToList Nil = []
myListToList (Elem x rest) = (x : (myListToList rest))
```

Aufgabenblatt 1

Ein möglicher Lösungsweg für das 1. Aufgabenblatt:

```
  -- 1.1
convert :: Integer -> [Integer]
convert x = reverse (convert' x)
  where
  convert' :: Integer -> [Integer]
  convert' x
      | x < 10 = [x]
      | otherwise = (x `mod` 10) : (convert' (x `div` 10))
  -- 1.2
quersumme :: Integer -> Integer
quersumme x = mysum (convert x)

mysum :: [Integer] -> Integer
mysum [] = 0
mysum (x : xs) = x + (mysum xs)

  -- using foldl power
quersumme2 :: Integer -> Integer
quersumme2 x = foldl (+) 0 (convert x)

  -- 1.3
dreiTeilbar :: Integer -> Bool
dreiTeilbar x = (quersumme x `mod` 3) == 0

  -- 1.4
sechsTeilbar :: Integer -> Bool
sechsTeilbar x = dreiTeilbar x && ((x `mod` 2) == 0)

  -- 1.5
elfTeilbar :: Integer -> Bool
elfTeilbar x = ((a (convert x)) `mod` 11) == 0
  where
  a :: [Integer] -> Integer
  a [] = 0
  a (x : []) = x
  a (x : y : xs) = x - y + a xs
```

Aufgabenblatt 2

Ein möglicher Lösungsweg für das 2. Aufgabenblatt:

```
  -- 2.1
diffFolge :: (Integer, Integer) -> [Integer]
```

```
diffFolge (m, n)
    | m ⩽ 0 = [m]
    | otherwise = m : (diffFolge (m − n, n))
  -- 2.2
teilt :: (Integer, Integer) → Bool
teilt x = head (reverse (diffFolge x)) ≡ 0
  -- 2.3
zahlenBlock :: Integer → [Integer]
zahlenBlock x
    | x < 1000 = [x]
    | otherwise = block : (zahlenBlock next)
      where
      block = x `mod` 1000
      next = x `div` 1000
  -- 2.4
zeichenreihenBlock :: Integer → [String]
zeichenreihenBlock zahl = [myshow y | y ← (zahlenBlock zahl)]
myshow :: Integer → String
myshow x = reverse (take 3 fillWithNull)
  where
  str :: String
  str = myshow' (fromInteger x)
  fillWithNull :: String
  fillWithNull = str ++ "000"
  myshow' :: Int → String
  myshow' x
      | x > 10 = (chr akt) : (myshow' next)
      | otherwise = [chr akt]
    where
    akt :: Int
    akt = 48 + (x `mod` 10)
    next :: Int
    next = x `div` 10
  -- 2.5
siebenTeilbar :: Integer → Bool
siebenTeilbar x = teilt (abs (as (zahlenBlock x)), 7)
as :: [Integer] → Integer
as [ ] = 0
as (x : [ ]) = x
as (x : y : xs) = x − y + (as xs)
```