Universität Siegen
Lehrstuhl Theoretische Informatik
Carl Philipp Reh

Übungsblatt 4

Aufgabe 1. Betrachten Sie folgende Definition von Bäumen:

```
data Tree a = Node a [Tree a]
```

(a) Implementieren Sie `Foldable` für `Tree`.

Lösung. Wir verwenden unsere schon implementierte Funktion

```
preorder :: Tree a -> [a]
```

und können damit `foldr` implementieren über

```
instance Foldable Tree where
  foldr f x t = foldr f x (preorder t)
```

(b) Implementieren Sie ein geeignetes Monoid zum Summieren.

Lösung.

```
data Plus = Add Int
instance Semigroup Plus where
  (<>) x y = case x of
               Add x' -> case y of
                 Add y' -> Add (x' + y')
instance Monoid Plus where
  mempty = Add 0
```

(c) Summieren Sie alle Werte eines Baums auf, indem Sie `foldMap` und das soeben definierte Monoid zum Summieren verwenden.

Lösung. Wir definieren zunächst eine Funktion, die den `Int` aus einem `Plus` wieder extrahiert:

```
fromPlus :: Plus -> Int
fromPlus x = case x of Add x' -> x'
```

Damit können wir dann schreiben:

```
sumTree' :: Tree Int -> Int
sumTree' t = fromPlus (foldMap Add t)
```

Aufgabe 2. Bei `Foldable` hat man die Wahl, `foldMap` oder `foldr` zu implementieren.

(a) Implementieren Sie `foldMap` über `foldr`, was wir `foldMap'` nennen.

Lösung.

```
foldMap' :: (Foldable t, Monoid m) =>
            (a -> m) -> t a -> m
foldMap' f x =
  foldr (\e y -> (f e) <> y) mempty x
```

(b) Implementieren Sie `foldr` über `foldMap`, was wir `foldr'` nennen. Welches Monoid müssen Sie hier verwenden?

Lösung. Das Monoid, was wir hier verwenden, ist Funktionskomposition mit der Identitätsfunktion. Dieses Monoid gibt es in Haskell unter dem Namen `Endo`. Der Deutlichkeit halber implementieren wir dies selbst:

```
data Endo a = MkEndo (a -> a)

instance Semigroup (Endo a) where
  (<>) f g = case f of
               MkEndo f' -> case g of
                 MkEndo g' -> MkEndo (f' . g')

instance Monoid (Endo a) where
  mempty = MkEndo id

fromEndo :: Endo a -> a -> a
fromEndo x = case x of MkEndo f -> f
```

Wir können dann schreiben

```
foldr' :: Foldable t =>
          (a -> b -> b) -> b -> t a -> b
foldr' f x l =
  fromEndo (foldMap (\e -> MkEndo (f e)) l) x
```

Erklärung: Die Funktion `f :: a -> b -> b` wird in einer bestimmten Reihenfolge auf jedes Element vom Typ `a`, das in `l` vorkommt, angewandt. Dies liefert eine Liste von Funktionen

```
[f_1, ..., f_n] :: [b -> b]
```

wobei $n \geq 0$, d.h. die Liste kann auch leer sein. Die Funktionen werden dann alle verknüpft:

```
r = f_1 . ... . f_n . id
```

Hierbei ist `.` Funktionskomposition und `id` die Identität. Wir benötigen `id` für den Fall, dass $n = 0$. Zuletzt wird dann `x` an `r` übergeben.

Wir betrachten als Beispiel `foldr (+) 0 [1,2]`: Wir erhalten die Funktionen `f_1 x = 1 + x` und `f_2 x = 2 + x`. Verknüpft gibt dies die Funktion `r x = 1 + (2 + x)`. Angewandt auf `0` erhalten wir `1 + (2 + 0) = 3`.

Aufgabe 3. Implementieren Sie folgendes Programm: Fragen Sie den Benutzer nach seinem Alter und geben Sie aus, ob dieser volljährig ist. Achten Sie darauf, dass der Benutzer kein negatives Alter eingibt.

Lösung. Wie zuvor definieren wir `stringToInt` über `readMaybe`:

```
import Text.Read(readMaybe)

stringToInt :: String -> Maybe Int
stringToInt = readMaybe
```

Die Funktion `response` implementieren wir getrennt von `IO`:

```
response :: String -> String
response x = case stringToInt x of
  Nothing -> "Cannot␣convert␣to␣string␣" ++ x
  Just i -> if i < 0
            then "Age␣must␣be␣>=␣0"
            else if i >= 18
              then "You␣are␣an␣adult"
              else "You␣are␣not␣an␣adult"
```

Die `main`-Funktion benutzt dann die `IO`-Monade für die Ein- und Ausgabe.

```
main :: IO ()
main = do
  r <- putStrLn "Please␣enter␣your␣age"
  x <- getLine
  putStrLn (response x)
```