6. Übung zur Vorlesung "Fortgeschrittene funktionale Programmierung"

Typklassen

Boris Dudelsack

11. November 2016

Aufgabe 1: Typklasse Appendable

```haskell
class Appendable a where
  (<>) :: a → a → a
  empty :: a

instance Appendable [a] where
  (<>) (x:xs) xs' = x : (xs <> xs')
  (<>) [] xs' = xs'
  empty = []

foldMapList :: Appendable m ⇒ (a → m) → [a] → m
foldMapList f = foldr ((<>) . f) empty

mcombine :: Appendable m ⇒ [m] → m
mcombine = foldMapList id

concat' :: Appendable a ⇒ [[a]] → [a]
concat' = map mcombine
```

Aufgabe 2: Appendable-Instanzen

```haskell
data Any = Any Bool
  deriving Show

data All = All Bool
  deriving Show

instance Appendable Any where
  empty = Any False
  (<>) (Any a) (Any b) = Any (a || b)

instance Appendable All where
  empty = All True
  (<>) (All a) (All b) = All (a && b)

any :: (a → Bool) → [a] → Bool
any f xs = resultAny (foldMapList (Any . f) xs)

all :: (a → Bool) → [a] → Bool
all f xs = resultAll (foldMapList (All . f) xs)

resultAny :: Any → Bool
resultAny (Any b) = b

resultAll :: All → Bool
```

```haskell
resultAll (All b) = b

data Sum a = Sum a
  deriving Show

data Product a = Product a
  deriving Show

instance Num a ⇒ Appendable (Sum a) where
  (<>) (Sum x) (Sum y) = Sum (x + y)
  empty = Sum 0

instance Num a ⇒ Appendable (Product a) where
  (<>) (Product x) (Product y) = Product (x * y)
  empty = Product 1

sum :: [Int] → Int
sum xs = resultSum (foldMapList Sum xs)

resultSum :: Sum Int → Int
resultSum (Sum x) = x

product :: [Int] → Int
product xs = resultProduct (foldMapList Product xs)

resultProduct :: Product Int → Int
resultProduct (Product x) = x
```

Aufgabe 3: Appendable für Bäume

```haskell
data Tree a = Empty | Node (Tree a) a (Tree a)

foldMapTree :: Appendable m ⇒ (a → m) → Tree a → m
foldMapTree _ Empty = empty
foldMapTree f (Node x y z) = foldMapTree f x <> f y <> foldMapTree f z

toList :: Tree a → [a]
toList = foldMapTree (: [])
```