Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2016/2017

Übungen zu Funktionaler Programmierung
Übungsblatt 8

Ausgabe: 2.12.2016, **Abgabe:** 9.12.2016 - 12:00 Uhr

Aufgabe 8.1 (3 Punkte) Schreiben Sie eine Klasse für eine überladene Additionsfunktion. Instanzieren Sie die Klasse für `Nat` und `PosNat`.

Lösungsvorschlag

```
class Addition a where
  add :: a -> a -> a

instance Addition Nat where
  add Zero n     = n
  add (Succ n) m = Succ (add n m)

instance Addition PosNat where
  add One n       = Succ' n
  add (Succ' n) m = Succ' (add n m)
```

Aufgabe 8.2 (6 Punkte)

1. Schreiben Sie eine Instanz der Klasse `Eq` für den Typ `Nat`.

2. Schreiben Sie eine Instanz der Klasse `Ord` für den Typ `Nat`. Es ist ausreichend den Operator (<=) zu definieren.

3. Schreiben Sie eine Instanz der Klasse `Enum` für den Typ `Nat`. Es ist ausreichend die Funktionen `toEnum` und `fromEnum` zu definieren.
   Beispiel:
   `take 3 $ map fromEnum [Zero .. ]` $\rightsquigarrow$ `[0,1,2]`

4. Schreiben Sie eine Instanz der Klasse `Show` für den Typ `Nat`. Nehmen Sie an, die Klasse sei wie folgt definiert:

   ```
   class Show a where
     show :: a -> String
   ```

   Die Definition der show-Funktion soll Ausgaben ähnlich denen vom Typ `Int` geben:

   ```
   show Zero ~> "0"
   show (Succ Zero) ~> "1"
   show (Succ (Succ (Succ Zero))) ~> "3"
   ```

5. Schreiben Sie eine Instanz der Klasse `Num` für den Typ `Nat`. Nutzen Sie folgende Vorgabe:

```
instance Num Nat where
  negate = undefined
  abs n  = n
  signum Zero = Zero
  signum n    = Succ Zero
  fromInteger = toEnum . fromInteger
```

Sie müssen lediglich die fehlenden Operatoren `(+)` und `(*)` definieren (Stichwort: Peano-Axiome).

6. Ändern Sie den Typ der unendlichen Liste `solutions` in `[(Nat,Nat,Nat)]`. Die Lösung soll auf Aufgabe 6.1.2 basieren, und weiterhin als Listenkomprehension definiert werden. Die Funktion `exp2store` wird nicht benötigt. Sie dürfen dabei alle zuvor definierten Klasseninstanzen nutzen.

Lösungsvorschlag

```
instance Eq Nat where
  Zero == Zero     = True
  Succ n == Succ m = n == m
  _ == _           = False

instance Ord Nat where
  Succ n <= Succ m = n <= m
  Zero <= _        = True
  _ <= _           = False

instance Enum Nat where
  toEnum 0           = Zero
  toEnum n | n > 0   = Succ (toEnum (n - 1))
  fromEnum Zero      = 0
  fromEnum (Succ n)  = fromEnum n + 1

instance Show Nat where
  show = show . fromEnum

instance Num Nat where
  Zero + n     = n
  (Succ n) + m = Succ (n + m)
  Zero * n     = Zero
  (Succ n) * m = m + (n * m)
  negate = undefined
  abs n  = n
  signum Zero = Zero
  signum n    = Succ Zero
  fromInteger = toEnum . fromInteger

solutions :: [(Nat,Nat,Nat)]
solutions = [ (x,y,z)
  | z <- [0..] , y <- [0..z], x <- [0..z], 3*x^2 + 2*y + 1 == z ]
```

Aufgabe 8.3 (3 Punkte) Definieren Sie folgende Haskell-Funktionen:

1. Symmetrische Differenz: $A \Delta B$.

2. `height :: Bintree a -> Int` berechnet die Höhe eines binären Baumes.

3. `isBalanced :: Bintree a -> Bool` testet, ob ein binärer Baum ausbalanciert ist.

Lösungsvorschlag

```
symDiff :: Eq a => [a] -> [a] -> [a]
symDiff a b = (a `diff` b) `union` (b `diff` a)
-- oder: symDiff a b = (a `union` b) `diff` (a `meet` b)

height :: Bintree a -> Int
height (Fork _ l r) = 1 + max (height l) (height r)
height Empty = 0

isBalanced :: Bintree a -> Bool
isBalanced (Fork _ l r) = (difference <= 1)
  && isBalanced l
  && isBalanced r
  where difference = abs (height l - height r)
isBalanced Empty = True
```