technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

# Übungen zu Funktionaler Programmierung
# Übungsblatt 7

**Ausgabe:** 1.12.2017, **Abgabe:** 8.12.2017 – 16:00 Uhr, **Block:** 4

**Aufgabe 7.1** (6 Punkte)

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

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

c) 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 .. ] ⇝ [0,1,2]`

d) Schreiben Sie eine Instanz der Klasse `Show` für den Typ `Nat`. Die Ausgabe soll sich ähnlich wie vom Typ `Int` verhalten:

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

e) 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). Sie dürfen nur die beiden Operatoren `(+)` und `(*)` benutzen. Weitere Hilfsfunktionen sind nicht erlaubt.

f) Ändern Sie den Typ der Liste `solutions` in `[(Nat,Nat,Nat)]` und passen Sie die Definition entsprechend an.

```
solutions :: [(Int,Int,Int)]
solutions = [ (x,y,z)
  | z <- [0..] , y <- [0..z^2], x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2 ]
```

**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
  showsPrec _ n = shows (fromEnum n)

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^2], x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2 ]
```

**Aufgabe 7.2** (3 Punkte) *Ausgabe*
Schreiben Sie eine Instanz der Klasse `Show` für den folgenden Datentyp für nichtleere binäre Bäume:

```
data STree a = BinS (STree a) a (STree a) | LeftS  (STree a) a
             | RightS  a (STree a)  | LeafS a
```

Die Ausgabe soll sich an den binären Bäumen aus der Vorlesung orientieren:
`BinS (LeftS (LeafS 9) 2) 4 (RightS 7 (LeafS 3))` $\leadsto$ `4(2(9,),7(,3))`

**Lösungsvorschlag**

```
instance Show a => Show (STree a) where
  showsPrec _ (LeafS a) = shows a
  showsPrec _ (LeftS l a)
    = shows a . showChar '(' . shows l . showString ",)"
  showsPrec _ (RightS a r)
    = shows a . showString "(," . shows r . showChar ')'
  showsPrec _ (BinS l a r) = shows a . showChar '('
    . shows l . showChar ',' . shows r . showChar ')'
```

**Aufgabe 7.3** (3 Punkte) *Bäume mit beliebigem Ausgrad*
Definieren Sie folgende Funktionen.

a) `treeAnd :: Tree Bool -> Bool` – Verhält sich wie `and`. Ergibt `True`, falls alle Knoten den Wert `True` enthalten.

b) `treeZip :: Tree a -> Tree b -> Tree (a,b)` – Eine Variante von `zip` für Bäume.

**Lösungsvorschlag**

```
treeAnd :: Tree Bool -> Bool
treeAnd (F b as) = b && all treeAnd as
treeAnd (V b) = b

treeZip :: Tree a -> Tree b -> Tree (a,b)
treeZip (F a as) (F b bs) = F (a,b) $ zipWith treeZip as bs
treeZip a b = V (root a,root b)
```