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 5

Ausgabe: 17.11.2017, **Abgabe:** 24.11.2017 – 16:00 Uhr, **Block:** 3

Aufgabe 5.1 (2 Punkte) *Unendliche Listen*
Schreiben Sie die angegebene Liste `solutions :: [(Int, Int, Int)]` so um, dass sie *alle* Lösungen der Gleichung $2x^3 + 5y + 2 = z^2$ enthält. Die Liste wird dadurch unendlich lang.

```
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z)
  | z <- [0..100]
  , y <- [0..100]
  , x <- [0..100]
  , 2*x^3 + 5*y + 2 == z^2
  ]
```

Lösungsvorschlag

Um eine Endlosschleife zu vermeiden, darf nur die erste Liste in der Komprehension unendlich sein. Da für $x, y > z$ keine Lösungen mehr existieren können, ist dies eine sinnvolle Einschränkung.

```
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
  ]
```

Aufgabe 5.2 (3 Punkte) *Zahlen als Datentypelemente*
Diese Aufgabe bezieht sich auf die in der Vorlesung vorgestellten rekursiven Datentypen `Nat`, `Int'` und `PosNat`.

a) Definieren Sie eine Konstante *drei* = 3 für den Datentyp `Int'` in Haskell.

b) Erweitern Sie die Datentypen für Zahlen um einen Datentyp für rationale Zahlen. Basieren Sie den Datentyp nur auf den Datentypen `Nat`, `Int'` und `PosNat`.

c) Definieren Sie eine Konstante $c = -\frac{3}{2}$ für Ihren Datentyp in Haskell.

Lösungsvorschlag

```
drei :: Int'
drei = Plus (Succ' (Succ' One))

data Rat = Rat Int' PosNat

c :: Rat
c = Rat (Minus (Succ' (Succ' One))) (Succ' One)
```

Aufgabe 5.3 (4 Punkte) *Rekursive Datentypen*
Definieren Sie folgende Haskell-Funktionen.

a) `natLength :: [a] -> Nat`, wie `length` für den Datentyp `Nat`.

b) `natDrop :: Nat -> [a] -> [a]`, wie `drop` für den Datentyp `Nat` anstatt `Int`.

c) `colistIndex :: Colist a -> Int -> a`, wie `(!!)` für `Colist a` anstatt `[a]`.

d) `streamTake :: Int -> Stream a -> [a]`, wie `take` für `Stream a` anstatt `[a]`.

Lösungsvorschlag

```
natLength :: [a] -> Nat
natLength (_:s) = Succ (natLength s)
natLength []    = Zero

natDrop :: Nat -> [a] -> [a]
natDrop (Succ n) (a:s) = natDrop n s
natDrop Zero     s     = s
natDrop _        []    = []

colistIndex :: Colist a -> Int -> a
colistIndex (Colist (Just (a,_))) 0         = a
colistIndex (Colist(Just (_,s))) n | n > 0 = colistIndex s (n-1)

streamTake :: Int -> Stream a -> [a]
streamTake 0 _ = []
streamTake n (a :< s) | n > 0 = a : streamTake (n-1) s
```

Aufgabe 5.4 (3 Punkte) *Modellierung*
Gegeben seien folgende Datentypen:

```
type ID = Int
data Bank = Bank [(ID,Account)] deriving Show
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show
```

Definieren Sie folgende Funktionen.

a) `credit :: Int -> ID -> Bank -> Bank` – Addiert den angegebenen Betrag auf das angegebene Konto.

b) `debit :: Int -> ID -> Bank -> Bank` – Subtrahiert den angegebenen Betrag von dem angegebenen Konto.

c) `transfer :: Int -> ID -> ID -> Bank -> Bank` – Überweist den angegebenen Betrag vom ersten Konto auf das zweite.

Lösungsvorschlag

```
credit :: Int -> ID -> Bank -> Bank
credit amount id (Bank ls)
  = Bank (updRel ls id entry{ balance = oldBalance + amount})
  where
    Just entry = lookup id ls
    oldBalance = balance entry

debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)

transfer :: Int -> ID -> ID -> Bank -> Bank
transfer amount id1 id2 = debit amount id1 . credit amount id2
```