Universität Siegen
Lehrstuhl Theoretische Informatik
Carl Philipp Reh

Übungsblatt 2

Aufgabe 1. Gegeben sei der Typ `Nat` aus der Vorlesung, definiert als

```
data Nat = Zero | Succ Nat
```

(a) Implementieren Sie eine geeignete `Show`-Instanz für `Nat`.

Lösung.

```
instance Show Nat where
  show x = case x of
           Zero -> "Zero"
           Succ n -> "Succ␣(" ++ show n ++ ")"
```

(b) Implementieren Sie die Funktion

```
fromInt :: Int -> Nat
```

welche einen `Int` in `Nat` umwandelt. Dabei soll gelten, dass fromInt für negative Zahlen Zero liefert und für positive Zahlen $n \geq 0$ soll $\text{Succ}^n(\text{Zero})$ geliefert werden.

Lösung.

```
fromInt x =
  if x < 0 then Zero
  else if x == 0 then Zero
  else Succ (fromInt (x - 1))
```

(c) Implementieren Sie die Funktion

```
fromInt' :: Int -> Maybe Nat
```

welche im Gegensatz zu fromInt bei negativen Zahlen Nothing liefert.

Lösung.

```
fromInt' x = if x < 0 then Nothing
             else Just (fromInt x)
```

(d) Implementieren Sie die Funktion

```
add :: Nat -> Nat -> Nat
```

welche zwei `Nat` addiert. Für $x, y \in \mathbb{N}$ soll gelten, dass

$$\text{add}(\text{Succ}^x(\text{Zero}), \text{Succ}^y(\text{Zero})) = \text{Succ}^{x+y}(\text{Zero}).$$

Lösung.

```
add x y = case x of
  Zero -> y
  Succ x' -> Succ (add x' y)
```

(e) Implementieren Sie eine Funktion, die es Ihnen erlaubt, explizite Rekursion bei `Nat` zu vermeiden. Implementieren Sie anschließen `add`, was wir nun `add'` nennen, darüber.

Lösung.

```
foldNat :: (a -> a) -> a -> Nat -> a
foldNat f v n = case n of
  Zero -> v
  (Succ y) -> f (foldNat f v y)

add' :: Nat -> Nat -> Nat
add' x y = foldNat Succ y x
```

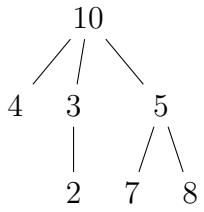
Ähnlich wie bei `foldr` für Listen bekommt `foldNat` drei Argumente:

- Die Funktion `f :: a -> a`, die wir für jedes `Succ` einmal anwenden.
- Den Anfangswert `v :: a`.
- Die natürliche Zahl `n :: Nat` selbst.

Die Funktion `foldNat` liefert dann für $n = \text{Succ}^z(\text{Zero})$, wobei $z \in \mathbb{N}$, den Wert $f^z(v)$. In der Implementierung von `add` wählen wir $f = \text{Succ}$, $v = \text{Succ}^y(\text{Zero})$ und $n = \text{Succ}^x(\text{Zero})$, wobei $x, y \in \mathbb{N}$. Damit erhalten wir, dass

$$\text{add'}(\text{Succ}^x(\text{Zero}), \text{Succ}^y(\text{Zero})) = \text{Succ}^x(\text{Succ}^y(\text{Zero})) = \text{Succ}^{x+y}(\text{Zero}).$$

Aufgabe 2. Betrachten Sie den folgenden Baum:



Dieser besteht aus einem Wurzelknoten, der mit 10 beschriftet ist. Der Wurzelknoten hat drei Kindknoten, welche mit 4, 3 und 5 beschriftet sind, usw.

(a) Definieren Sie eine geeignete Data-Deklaration für Bäume. Es soll möglich sein, diese auch mit anderen Beschriftungen als Werte vom Typ `Int` zu versehen. Beschreiben Sie, welche Typkonstruktoren und Wertkonstruktoren Sie definieren.

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

Dies definiert den Typkonstruktor `Tree :: * -> *` und den Wertkonstruktor `Node :: a -> [Tree a] -> Tree a`.

(b) Implementieren Sie den obigen Baum.

Lösung.

```
t :: Tree Int
t = Node 10 [Node 4 []
             , Node 3 [Node 2 []]
             , Node 5 [Node 7 [], Node 8 []]]
```

(c) Implementieren Sie eine geeignete `Show`-Instanz für Ihre Bäume.

Lösung.

```
instance Show a => Show (Tree a) where
  show t = case t of
    (Node x l) -> "(" ++ show x
                  ++ "," ++ show l ++ ")"
```

Wir verwenden in unserer Implementierung `show x`, wobei `x :: a`, was `show` auf die Beschriftung des Wurzelknoten anwendet. Damit dies zur Verfügung steht, müssen wir `Show a =>` an den Anfang unserer Instanz-Deklaration schreiben. Natürlich muss dann auch der Elementtyp des Baums `Show` implementieren. Wir verwenden ebenfalls `show l`, wobei `l :: [Tree a]`. Dies ruft die `Show`-Instanz von Listen auf, die wiederum `show` für unsere Bäume aufruft.

(d) Implementieren Sie die Funktion

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

welche die *Preorder* eines Baums als Liste liefert. Die Preorder vom Beispielbaum wäre $10, 4, 3, 2, 5, 7, 8$.

Lösung.

```
preorder t = case t of
  (Node x l) -> (x : concat (map preorder l))
```

Die Funktion `concat :: [[a]] -> [a]` nimmt eine Liste von Listen und konkateniert all ihre Elemente. Zum Beispiel gilt

```
concat [[1,2],[3]] = [1,2,3]
```

Die Funktion `map :: (a -> b) -> [a] -> [b]` wendet eine Funktion auf jedes Element einer Liste an. Zum Beispiel gilt

```
map show [1,2] = ["1","2"]
```

(e) Implementieren Sie die Funktion

```
sumTree :: Tree Int -> Int
```

welche die Werte eines Baums von `Int` aufsummiert.

Lösung. `sumTree t = foldr (+) 0 (preorder t)`