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

Übungen zu Funktionaler Programmierung
Übungsblatt 9

Ausgabe: 15.12.2017, **Abgabe:** 22.12.2017 – 16:00 Uhr, **Block:** 5

Aufgabe 9.1 (4 Punkte)

a) Zeigen Sie, dass die Rekursionsgleichung `fib` eine Funktion definiert. Definieren Sie dazu eine Schrittfunktion Φ analog zu der auf Folie 146.

b) Beweisen Sie durch Induktion, dass lfp(Φ) keine natürliche Zahl auf $\perp$ abbildet.

Lösungsvorschlag

a)

$$\begin{aligned}\Phi : (\mathbb{N} \to \mathbb{N}_\perp) &\to (\mathbb{N} \to \mathbb{N}_\perp)\\ f &\mapsto \lambda n.\text{if } n > 1 \text{ then } f(n-1) + f(n-2) \text{ else } 1\end{aligned}$$

Φ ist stetig, wenn man die Addition zur *strikten* Funktion auf $\mathbb{N}_\perp$ erweitert, d. h. $n+\perp$ und $\perp+n$ auf $\perp$ setzt.

b) Induktionsvoraussetzung: $\text{lfp}(\Phi)(n) \neq \perp$ und $\text{lfp}(\Phi)(m) \neq \perp$ für alle $m < n$.
Induktionsanfang:

$$\begin{aligned}&\text{lfp}(\Phi)(0)\\ =&\Phi(\text{lfp}(\Phi))(0)\\ =&(\lambda n.\text{if } n > 1 \text{ then lfp}(\Phi)(n-1) + \text{lfp}(\Phi)(n-2) \text{ else } 1)(0)\\ =&1 \neq \perp\end{aligned}$$

$$\begin{aligned}&\text{lfp}(\Phi)(1)\\ =&\Phi(\text{lfp}(\Phi))(1)\\ =&(\lambda n.\text{if } n > 1 \text{ then lfp}(\Phi)(n-1) + \text{lfp}(\Phi)(n-2) \text{ else } 1)(1)\\ =&1 \neq \perp\end{aligned}$$

Induktionsschritt:

$$\begin{aligned}&\text{lfp}(\Phi)(n+1)\\ =&\Phi(\text{lfp}(\Phi))(n+1)\\ =&(\lambda n.\text{if } n > 1 \text{ then lfp}(\Phi)(n-1) + \text{lfp}(\Phi)(n-2) \text{ else } 1)(n+1)\\ =&\text{lfp}(\Phi)(n) + \text{lfp}(\Phi)(n-1)\\ &(\mathit{Induktionsvoraussetzung})\\ \neq&\perp\end{aligned}$$

Aufgabe 9.2 (3 Punkte)
Definieren Sie folgende Haskell-Funktionen.

a) `isCyclic :: Eq a => Graph a -> Bool` – Erkennt, ob ein Graph zyklisch ist. Sie können hier den transitiven Abschluss nutzen.
Beispiele:
`isCyclic graph1` ⇝ `True`
`isCyclic graph2` ⇝ `False`

b) `depthFirst :: Eq a => a -> Graph a -> [a]` – Ähnlich wie `preorder` und `postorder` auf Bäumen, sollen die Knoten des Graphen in einer bestimmten Reihenfolge als Liste ausgegeben werden. Die Funktion erhält einen Startknoten und gibt dann weitere Knoten durch Tiefensuche aus.

Lösungsvorschlag

```
isCyclic :: Eq a => Graph a -> Bool
isCyclic graph = any self nodes where
  G nodes closure = closureF graph
  self node = node `elem` closure node

depthFirst :: Eq a => a -> Graph a -> [a]
depthFirst start (G _ adj) = df [start] [] where
  df (a:as) visited
    | a `elem` visited = df as visited
    | otherwise        = df (adj a ++ as) (visited ++ [a])
  df [] visited        = visited
```

Aufgabe 9.3 (2 Punkte) *Kinds*
Bestimmen Sie den Kind folgender Typkonstruktoren.

a)
```
class C f where
  comp :: f b c -> f a b -> f a c
```
Bestimmen Sie den Kind von `f`.

b) `data T f g = T (f String Int) (g Bool)`
Bestimmen Sie den Kind von `T`.

Lösungsvorschlag

a) `f :: * -> * -> *`

b) `T :: (* -> * -> *) -> (* -> *) -> *`

Aufgabe 9.4 (3 Punkte) *Typfamilien*
Gegeben sei folgende Typklasse:

```
class Listable l where
  type Item l :: *
  toList :: l -> [Item l]
```

Die Funktion `toList` wandelt eine Eingabe in eine Liste um. Überladen Sie die Funktion für die angegebenen Typen.

a) `Colist a` – Gibt die zugehörige Liste aus.

b) `data Map a b = Map [(a,b)]` – Ein Datentyp für eine Assoziationsliste (siehe Folie 53). Es sollen in der Liste nur die Werte ausgegeben werden. Die Argumente bzw. Schlüssel entfallen.

c) `Nat` – Die bijektive Funktion der Isomorphie von `Nat` und `[()]`.

Hinweis: Die Typklasse macht gebrauch von einer Typfamilie. Um Typfamilien zu nutzen, muss die Spracherweiterung *TypeFamilies* aktiviert werden. Fügen Sie das Pragma

```
{-# LANGUAGE TypeFamilies #-}
```

an den Kopf Ihrer Haskell-Datei ein.

Lösungsvorschlag

```
instance Listable (Colist a) where
  type Item (Colist a) = a
  toList ls = case split ls of
    Just (a,as) -> a : toList as
    Nothing -> []

instance Listable (Map a b) where
  type Item (Map a b) = b
  toList (Map ((a,b):as)) = b : toList (Map as)
  toList (Map [])         = []

instance Listable Nat where
  type Item Nat = ()
  toList Zero     = []
  toList (Succ n) = ():toList n
```