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

Übungen zu Funktionaler Programmierung
Übungsblatt 11

Ausgabe: 13.1.2016, **Abgabe:** 20.1.2017 - 12:00 Uhr

Aufgabe 11.1 (4 Punkte)

1. Zeigen Sie, dass die Rekursionsgleichung `fib` eine Funktion definiert. Definieren Sie dazu eine Schrittfunktion Φ analog zu der auf Folie 127.

2. Beweisen Sie durch Induktion, dass $\mathrm{lfp}(\Phi)$ keine natürliche Zahl auf $\perp$ abbildet.

Lösungsvorschlag

1.

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

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

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

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

   Induktionsschritt:

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

Aufgabe 11.2 (4 Punkte) Definieren Sie folgende Haskell-Funktionen.

1. `isCyclic :: Eq a => Graph a -> Bool` erkennt, ob ein Graph zyklisch ist. Sie können hier den transitiven Abschluss nutzen.
   Beispiele:
   `isCyclic graph1` $\rightsquigarrow$ `True`
   `isCyclic graph2` $\rightsquigarrow$ `False`

2. `undirected :: Eq a => Graph a -> Graph a` macht aus einem Graphen einen ungerichteten Graphen. Ein ungerichteter Graph lässt sich als gerichteter Graph darstellen, indem für jede Kante eine Kante in die Gegenrichtung eingefügt wird.
   Beispiel: `undirected graph1` $\rightsquigarrow$
   1 -> [2,3,4]; 2 -> [1,6]; 3 -> [1,4,6,5]; 4 -> [1,3,6]; 5 -> [3,5,6]; 6 -> [2,4,5,3]

Lösungsvorschlag

```
isCyclic :: Eq a => Graph a -> Bool
isCyclic graph = or (map self nodes) where
  G nodes closure = closureF graph
  self node = node `elem` closure node

undirected :: Eq a => Graph a -> Graph a
undirected (G nodes sucs) = G nodes sucs' where
  sucs' n = sucs n `union` [n' | n' <- nodes, n `elem` sucs n']
```

Aufgabe 11.3 (4 Punkte) Definieren Sie folgende partielle Funktionen mithilfe des Maybe-Datentyps.

1. `safeDiv :: Int -> Int -> Maybe Int` berechnet die Division zweier Ganzzahlen. Berücksichtigt jedoch die Teilung durch 0.

2. `safeSqrt :: Int -> Maybe Int` berechnet die Wurzel für positive Ganzahlen.
   Hilfsfunktion:

   ```
   intsqrt :: Int -> Int
   intsqrt = floor . sqrt . fromIntegral
   ```

3. `f :: Int -> Int -> Int -> Maybe Int` berechnet folgende Funktion:

$$f(x, y, z) = \frac{\sqrt{x}}{\sqrt{\frac{y}{z}}}$$

   Lösen Sie die Aufgabe mit den Funktionen `safeDiv` und `safeSqrt`. Sie können die Funktionen aus dem Modul `Data.Maybe` zu Hilfe nehmen. Dazu müssen Sie die Zeile „`import Data.Maybe`“ in Ihre .hs-Datei einfügen. Detaillierte Informationen zu `Data.Maybe` finden Sie mit Hoogle (https://www.haskell.org/hoogle/).

Lösungsvorschlag

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

safeSqrt :: Int -> Maybe Int
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (intsqrt x)
```

```haskell
f :: Int -> Int -> Int -> Maybe Int
f x y z = if isNothing par1 || isNothing par2
  then Nothing
  else safeDiv (fromJust par1) (fromJust par2)
  where
    par1 = safeSqrt x
    par2 = maybe Nothing safeSqrt (safeDiv y z)
```