

Altklausur Bry 2012 Remastered für Programmierung und Modellierung 2016

Alexander Isenko

July 17, 2016

Besprechung am 22. Juli 2016

Aufgabe 1

Hier wird nach Funktionen gefragt die ihr euch selber einfallen lassen könnt, total egal wie kompliziert oder einfach sie sind, sie müssen lediglich den Bedingungen entsprechen.

- a) Definieren sie eine monomorphe Funktion (keine Typvariablen, nur feste Typen)

Lösung:

```
1 aufgabel1 :: Int -> Int
2 aufgabel1 x = x + 1
```

- b) Definieren sie eine polymorphe Funktion (mit Typvariablen)

- i) parametrisch polymorph (keine Typvariablen vor ($=>$))

```
1 aufgabel2i :: a -> a
2 aufgabel2i x = x
```

- ii) ad-hoc polymorph (mid. eine Typvariable vor ($=>$))

```
1 aufgabel2ii :: Num a => a -> a
2 aufgabel2ii x = x + x
```

- c) Definieren sie eine *gecurrierte* Funktion (eine Funktion die ein Tupel nimmt, aber mit `curry` stattdessen die Argumente hintereinander akzeptiert)

```
1  -- Hilfestellung:
2
3  > :t curry
4  curry :: ((a, b) -> c) -> a -> b -> c
```

Lösung:

```
1  aufgabe1c :: a -> b -> a
2  aufgabe1c a b = curry f a b
3      where
4          f :: (a,b) -> a
5          f (a,b) = a
```

Aufgabe 3

Definieren sie die Funktion `reverse` für Listen in Haskell.

```
1      > :t reverse
2      [a] -> [a]
3
4      > reverse "hallo"
5      "ollah"
6
7      > reverse [1,2,3]
8      [3,2,1]
9
10     > reverse []
11     []
```

Lösung:

```
1  aufgabe3 :: [a] -> [a]
2  aufgabe3 []      = []
3  aufgabe3 (x:xs) = aufgabe3 xs ++ [x]
```

Aufgabe 4

Sei folgender Code gegeben, was ist das Ergebnis von `res`?

```
1 y = 5
2 x = 2
3 goo y      = x * y
4 fuu (x,y) = x + goo y
5 res = (goo y, fuu (5,7))
```

Lösung:

```
1      (goo y, fuu (5,7))      -- y = 5
2  => (goo 5, fuu (5,7))      -- goo auswerten
3  => (x * 5, fuu (5,7))      -- x = 2
4  => (2 * 5, fuu (5,7))      -- 2 * 5 = 10
5  => (10, fuu (5,7))          -- fuu auswerten
6  => (10, 5 + goo 7)          -- goo auswerten
7  => (10, 5 + x * 7)          -- x = 2
8  => (10, 5 + 2 * 7)          -- 5 + 2 * 7 = 19
9  => (10, 19)
```

Aufgabe 6

Definieren sie das Standardvektorkreuzprodukt mithilfe von `zip` und `Data.List.foldl/foldr`

```
1  -- Typsignatur:
2  --
3  skalarProdukt :: Num a => [a] -> [a] -> a
4  --
5  -- Beispiel:
6  --
7  skalarProdukt [1,2,3] [4,5,6] = 1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32
8
9  -- Hilfestellung:
10 --
11 zip :: [a] -> [b] -> [(a,b)]
12 zip [] [] = []
13 zip (x:xs) (y:ys) = (x,y) : zip xs ys
14
15 foldl :: (b -> a -> b) -> b -> [a] -> b
16 foldl f acc [] = acc
17 foldl f acc (x:xs) = foldl f (f acc x) xs
18
19 foldr :: (a -> b -> b) -> b -> [a] -> b
20 foldr f acc [] = acc
21 foldr f acc (x:xs) = f x (foldr f acc xs)
```

Lösung:

```
1  -- Da Addition kommutativ ist, können wir foldl
2  -- oder foldl benutzen
3  --
4  skalarProdukt :: Num a => [a] -> [a] -> a
5  skalarProdukt xs ys =
6      foldl (\acc (x,y) -> x*y + acc) 0 (zip xs ys)
7
8  skalarProdukt' :: Num a => [a] -> [a] -> a
9  skalarProdukt' xs ys =
10     foldr (\(x,y) acc -> x*y + acc) 0 (zip xs ys)
```

Aufgabe 7

Sei folgender Code gegeben.

```
1  ks = [3,5,7]
2
3  pred = \x -> x < 0
4
5  f p x (y:ys) = if p y
6                  then f (not . p) x ys
7                  else f p      x ys
8  f p x _      = p x
9
10 g x = let f x = x
11       in g (f x)
```

Geben sie wenn möglich den Wert den Typ der folgenden Ausdrücke an:

a) f

Lösung:

```
1  -- Aus Z.5 sieht man dass (p y) ein Bool zurückgibt,  
2  -- weil es im Prädikat des if-then-else steht  
3  p :: a -> Bool  
4  
5  -- 2. Aus Z.8 sehe ich dass die Funktion f als  
6  -- Rückgabewert ein Bool hat, weil wir (p x) aufrufen  
7  f :: (a -> Bool) -> ... -> ... -> Bool  
8  
9  -- 3. Aus Z.5 und Z.8 sieht man dass in 'p'  
10 -- sowohl 'x' als auch 'y' eingesetzt werden  
11 -- => sie sind vom gleichen Typ  
12  
13 -- 4. Wir sehen aus Z.5 dass (y:ys) eine Liste ist.  
14 f :: (a -> Bool) -> a -> [a] -> Bool
```

b) f pred 0 []

Lösung:

```
1  -- Da für 'f' alle Argumente befüllt sind ist der  
2  -- Rückgabewert  
3  Bool  
4  -- Das dritte Argument ist eine leere Liste,  
5  -- wir kommen in den zweiten Fall von 'f', Z.8  
6  pred 0 ==> 0 < 0 ==> False
```

c) f pred 0 ks

Lösung:

```
1  -- Da für 'f' alle Argumente befüllt sind ist der
2  -- Rückgabewert
3  Bool
4  -- 2. Das dritte Argument ist keine leere Liste,
5  -- wir kommen in den ersten Fall von 'f', Z.8
6
7  f pred 0 [3,5,7]
8  => if pred 3      -- (3 < 0) => False
9      then f (not . pred) 0 [5,7]
10     else f (pred) 0 [5,7]
11
12  f pred 0 [5,7]   -- Erster Fall von 'f',
13                  -- Liste ist nicht leer
14
15  => if pred 5      -- (5 < 0) => False
16      then f (not . pred) 0 [7]
17     else f (pred) 0 [7]
18
19  f pred 0 [7]     -- Erster Fall von 'f',
20                  -- Liste ist nicht leer
21
22  => if pred 7      -- (7 < 0) => False
23      then f (not . pred) 0 []
24     else f (pred) 0 []
25
26  f pred 0 []      -- Zweiter Fall von 'f',
27                  -- da Liste leer Z.8
28
29  => pred 0 => 0 < 0 => False
```

d) g

Lösung:

```
1  -- Wir sehen dass die Funktion 'f' die in 'g'
2  -- definiert wurde, lediglich ihr Argument zurückgibt
3  f :: a -> a    -- identisch zur Funktion id
4  id :: a -> a
5
6  -- Wir haben nur einen Fall, wo wir uns rekursiv
7  -- mit dem gleichen Argument immer wieder aufrufen.
8  -- Es gibt sonst keine Einschränkungen, der Typ ist
9  g :: a -> b
```

e) g ks

Lösung:

```
1  -- Da 'ks' eine Liste von Zahlen ist, ist sie vom Typ.
2  ks :: Num a => [a]
3  -- Da 'g' vom Typ 'a -> b' ist, setzen wir den Typ
4  -- von 'ks' für das Argument ein.
5  g ks :: Num a => [a] -> b
6  -- Diese Funktion terminiert nicht, somit
7  -- gibt es kein Ergebnis
```