Altklausur Bry 2012 Remastered für Programmierung und Modellierung 2016

Alexander Isenko

July 18, 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:

```haskell
aufgabe1a :: Int -> Int
aufgabe1a x = x + 1
```

b) Definieren sie eine polymorphe Funktion (mit Typvariablen)

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

```haskell
aufgabe1bi :: a -> a
aufgabe1bi x = x
```

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

```haskell
aufgabe1bii :: Num a => a -> a
aufgabe1bii x = x + x
```

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

```
-- Hilfestellung:

   > :t curry
   curry :: ((a, b) -> c) -> a -> b -> c
```

Lösung:

```
aufgabe1c :: a -> b -> a
aufgabe1c a b = curry f a b
    where
        f :: (a,b) -> a
        f (a,b) = a
```

Aufgabe 3

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

```
        > :t reverse
        [a] -> [a]

        > reverse "hallo"
        "ollah"

        > reverse [1,2,3]
        [3,2,1]

        > reverse []
        []
```

Lösung:

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

Aufgabe 4

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

```haskell
y = 5
x = 2
goo y     = x * y
fuu (x,y) = x + goo y
res = (goo y, fuu (5,7))
```

Lösung:

```haskell
   (goo y, fuu (5,7))     -- y = 5
=> (goo 5, fuu (5,7))     -- goo auswerten
=> (x * 5, fuu (5,7))     -- x = 2
=> (2 * 5, fuu (5,7))     -- 2 * 5 = 10
=> (10,    fuu (5,7))     -- fuu auswerten
=> (10,    5 + goo 7)     -- goo auswerten
=> (10,    5 + x * 7)     -- x = 2
=> (10,    5 + 2 * 7)     -- 5 + 2 * 7 = 19
=> (10, 19)
```

Aufgabe 6

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

```haskell
--  Typsignatur:
--
skalarProdukt :: Num a => [a] -> [a] -> a
--
--  Beispiel:
--
skalarProdukt [1,2,3] [4,5,6] = 1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32

-- Hilfestellung:
--
zip :: [a] -> [b] -> [(a,b)]
zip []     []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

Lösung:

```haskell
-- Da Addition kommutativ ist, können wir foldl
-- oder foldl benutzen
--
skalarProdukt :: Num a => [a] -> [a] -> a
skalarProdukt xs ys =
    foldl (\acc (x,y) -> x*y + acc) 0 (zip xs ys)

skalarProdukt' :: Num a => [a] -> [a] -> a
skalarProdukt' xs ys =
    foldr (\(x,y) acc  -> x*y + acc) 0 (zip xs ys)
```

Aufgabe 7

Sei folgender Code gegeben.

```haskell
ks = [3,5,7]

pred = \x -> x < 0

f p x (y:ys) = if p y
                 then f (not . p) x ys
                 else f p         x ys
f p x _      = p x

g x = let f x = x
      in g (f x)
```

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

a) `f`

Lösung:

```
-- Aus Z.5 sieht man dass (p y) ein Bool zurückgibt,
-- weil es im Prädikat des if-then-else steht
p :: a -> Bool

-- 2. Aus Z.8 sehe ich dass die Funktion f als
-- Rückgabewert ein Bool hat, weil wir (p x) aufrufen
f :: (a -> Bool) -> ... -> ... -> Bool

-- 3. Aus Z.5 und Z.8 sieht man dass in 'p'
-- sowohl 'x' als auch 'y' eingesetzt werden
-- => sie sind vom gleichen Typ

-- 4. Wir sehen aus Z.5 dass (y:ys) eine Liste ist.
f :: (a -> Bool) -> a -> [a] -> Bool
```

b) `f pred 0 []`

Lösung:

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

c) `f pred 0 ks`

Lösung:

```
-- Da für 'f' alle Argumente befüllt sind ist der
-- Typ vom Rückgabewert
Bool
-- 2. Das dritte Argument ist keine leere Liste,
-- wir kommen in den ersten Fall von 'f', Z.8

f pred 0 [3,5,7]
=> if pred 3      -- (3 < 0) => False
       then f (not . pred) 0 [5,7]
       else f (pred) 0 [5,7]

f pred 0 [5,7]     -- Erster Fall von 'f',
                   -- Liste ist nicht leer

=> if pred 5      -- (5 < 0) => False
      then f (not . pred) 0 [7]
      else f (pred) 0 [7]

f pred 0 [7]       -- Erster Fall von 'f',
                   -- Liste ist nicht leer

=> if pred 7       -- (7 < 0) => False
       then f (not . pred) 0 []
       else f (pred) 0 []

f pred 0 []        -- Zweiter Fall von 'f',
                   -- da Liste leer Z.8

=> pred 0 => 0 < 0 => False
```

d) `g`

Lösung:

```
-- Wir sehen dass die Funktion ’f’ die in ’g’
-- definiert wurde, lediglich ihr Argument zurückgibt
f :: a -> a     -- identisch zur Funktion id
id :: a -> a

-- Wir haben nur einen Fall, wo wir uns rekursiv
-- mit dem gleichen Argument immer wieder aufrufen.
-- Es gibt sonst keine Einschränkungen, der Typ ist
g :: a -> b
```

e) `g ks`

Lösung:

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