Übungen zu Ausdrücken - Programmierung und Modellierung 2016

Alexander Isenko

July 21, 2016

Besprechung am 22.Juli 2016

Aufgabe 1

Geben sie den **abstraktesten** Typ der folgenden Ausdrücke an.

a) `f = 1`

Lösung:

```
-- Wir fangen wie gewohnt an die Argumente bzw.
-- den Rückgabewert mit Typvariablen zu füllen
f :: a
f = 1

-- Hier haben wir nur einen Rückgabewert
-- Da es eine Zahl ist, muss es eine Instanz in
-- Num haben
f :: Num a => a
```

b) `f x = x + 1`

Lösung:

```
f :: a -> b
f x = x + 1
-- Diese Funktion nimmt ein Argument und darauf wird
-- (+) angewendet. Das ist eine Funktion aus Num
f :: Num a => a -> b
-- Wenn wir was addieren, muss es vom gleichen Typ
-- sein
f :: Num a => a -> a
```

c) `f = \x -> x`

Lösung:

```haskell
-- Das ist eine Lambda-Funktion die genauso
-- wie in b) ein Argument nimmt.
f :: a -> b
f = \x -> x
-- analog
f x = x
-- Wir können keine Aussagen über den genauen Typ
-- sagen, aber wir wissen das wir 'x' zurückgeben
f :: a -> a
```

d) `f y = \x -> x + y`

Lösung:

```haskell
-- man kann beide Schreibweisen auch verbinden
f :: a -> b -> c
f y = \x -> x + y

-- analog
f y x = x + y

-- Wir sehen dass wir beide Argumente addieren
f :: (Num a, Num b) => a -> b -> c

-- Addition braucht gleiche Typen
f :: Num a => a -> a -> c

-- Wir geben die addierten Werte zurück
f :: Num a => a -> a -> a
```

e) `f x y = compare x y`

Lösung:

```haskell
f :: a -> b -> c
f x y = compare x y

-- wir vergleichen beide Argumente
-- => sind von Ord
-- => gleicher Typ
f :: Ord a => a -> a -> c
f x y = compare x y

-- compare gibt einen sog. Ordering
-- Typ zurück (LT,EQ,GT)
f :: Ord a => a -> a -> Ordering
f x y = compare x y
```

f)
```haskell
f x y z
  | y         = 1
  | x == z    = x * 5
  | otherwise = z - 1
```

Lösung:

```haskell
-- Diese Funktion hat 3 Argumente
f :: a -> b -> c -> d
f x y z
  | y         = 1
  | x == z    = x * 5
  | otherwise = z - 1

-- 'y' wird in einem Guard benutzt
f :: a -> Bool -> c -> d

-- 'x' und 'z' werden verglichen
-- => sind gleich und sind von Eq
f :: Eq a => a -> Bool -> a -> d

-- 'x' und 'z' benutzen (*) und (-)
-- => sind von Num
f :: (Num a, Eq a) => a -> Bool -> a -> d

-- wir geben entweder 'x' oder 'z' zurück
f :: (Num a, Eq a) => a -> Bool -> a -> a
```

g) `f 'c' (y:ys) = []`

Lösung:

```haskell
-- Diese Funktion hat zwei Argumente
f :: a -> b -> c
f 'c' (y:ys) = []

-- 'c' ist Patternmatch auf ein Char
f :: Char -> b -> c

-- (y:ys) sagt dass es eine Liste ist
f :: Char -> [b] -> c

-- Der Rückgabewert ist eine leere Liste
-- die kann von beliebigen Typ sein
f :: Char -> [b] -> [c]
```

h) `f x y = [z | z <- y, mod x z == 1]`

Lösung:

```haskell
-- Diese Funktion nimmt zwei Argumente
f :: a -> b -> c
f x y = [z | z <- y, mod x z == 1]

-- wir sehen dass aus 'y' Elemente
-- rausgezogen werden (z <- y)
-- => y ist eine Liste
f :: a -> [b] -> c

-- wir benutzen modulo auf x und z
-- => beide von Integral
-- => gleicher Typ
f :: Integral a => a -> [a] -> c

-- Wir geben eine Liste zurück mit
-- Elementen aus 'y'
f :: Integral a => a -> [a] -> [a]
```

i)
```
f w x y z = let fooooo x
                  | w = z
                  | otherwise = y
            in fooooo 10
```

Lösung:

```
-- Diese Funktion nimmt vier Argumente
f :: a -> b -> c -> d -> e

-- wir sehen dass 'w' in einem Guard
-- benutzt wird
-- => w :: Bool
f :: Bool -> b -> c -> d -> e

-- foo nimmt ein Argument und gibt
-- z oder y zurück und wird mit 10
-- aufgerufen
-- => z und y vom gleichen Typ
-- => ist der Rückgabewert
f :: Bool -> b -> a -> a -> a
```

j)
```
f x y = case elem x y of
            True  -> 0
            False -> 1.0
```

Lösung:

```
-- Diese Funktion nimmt zwei Argumente
f :: a -> b -> c
f x y = case elem x y of
            True  -> 0
            False -> 1.0

-- wir sehen dass 'x' in 'y' mit
-- 'elem' gesucht wird
-- => y ist eine Liste
-- => x vom gleichen Typ wie y-Elemente
-- => wir müssen vergleichen können (Eq)
f :: Eq a => a -> [a] -> c

-- wir geben 1.0 zurück
-- => Rückgabewert ist Fractional
f :: (Eq a, Fractional c) => a -> [a] -> c
```

k) `f x = map (/2) [1..x]`

Lösung:

```haskell
-- Diese Funktion nimmt ein Argument
f :: a -> b
f x = map (/2) [1..x]

-- [1..x] bedeutet dass wir eine
-- Enumeration benutzen. Die Typklasse
-- heißt Enum
f :: Enum a => a -> b

-- wir wenden auf jedes Element der
-- Liste (/2) an. Für Divison brauchen
-- Fractional
f :: (Enum a, Fractional b) => a -> [b]

-- Wenn wir Dividieren wollen, mussen beide
-- Elemente vom gleichen Typ sein
f :: (Enum a, Fractional a) => a -> [a]
```

l) `f g = foldl g 0 [(-5.0), (-4.5)..(-0.5)]`

Lösung:

```
-- Diese Funktion nimmt ein Argument
f :: a -> b
f g = foldl g 0 [(-5.0), (-4.5)..(-0.5)]

-- zur Erinnerung, foldl(-eft) faltet
-- eine Liste von links mit einer
-- Funktion die das Element + Akkumulator
-- nimmt
foldl :: (b -> a -> b) -> b -> [a] -> b

-- die Liste von Zahlen besteht aus
-- einer Enumeration und Fließkommazahlen
[(-5.0), (-4.5)..(-0.5)] :: (Fractional a, Enum a) => [a]

-- Der Akkumulator 0 ist zunächst nur irgendeine Zahl
0 :: Num a => a

-- f erwartet von uns eine Funktion die etwas
-- mit dem Element + Akkumulator anstellt
g :: (b -> a -> b)
      |    |    |_______
      |    |            \
     akk  listelement    ergebnis

-- Die Liste besteht aus Fractional + Enum
g :: (Fractional a, Enum a) => (b -> a -> b)

-- Der Akkumulator ist von Num
g :: (Fractional a, Enum a, Num b) => (b -> a -> b)

-- Nun können wir das in f einsetzen
f :: (Fractional a, Enum a, Num b) => (b -> a -> b) -> b
```