TECHNISCHE UNIVERSITÄT DRESDEN

PROGRAMMIERUNG

ÜBUNG 4: FUNKTIONEN HÖHERER ORDNUNG

Eric Kunze

eric.kunze@tu-dresden.de

TU Dresden, 04. Mai 2022

letzte Änderung:
04.05.2022, 11:03

Typpolymorphie &
Funktionen höherer Ordnung

TYPPOLYMORPHIE

- **bisher**: Funktionen mit konkreten Datentypen
  z.B. `length :: [Int] -> Int`
- **Problem**: Funktion würde auch auf anderen Datentypen funktionieren
  z.B. `length :: [Bool] -> Int` oder `length :: String -> Int`
- **Lösung**: Typvariablen und polymorphe Funktionen
  z.B. `length :: [a] -> Int`

Bei konkreter Instanziierung wird Typvariable an entsprechenden Typbezeichner gebunden (z.B. `a = Int` oder `a = Bool`).

- Der Aufruf `length [1,5,2,7]` liefert für die Typvariable `a = Int`.
- Der Aufruf `length [True, False, True, True, False]` liefert die Belegung `a = Bool`.
- Der Aufruf `length "hello"` impliziert `a = Char`.

FUNKTIONEN

Wir kennen bereits einige Möglichkeiten Funktionen zu notieren. Hier seien einige weitere erwähnt.

anonyme Funktionen. Funktionen ohne konkreten Namen
: z.B. `(\x -> x+1)` ist die Addition mit 1

```
(\x -> x+1) 4 = 5
```

Operator $\leftrightarrow$ **Funktion** Aus Operatoren (wie z.B. +) kann man eine Funktion machen und vice versa.

- Operator $\rightarrow$ Funktion: Klammern

```
(+) :: Int -> Int -> Int
(+) x y =  x + y
```

- Funktion $\rightarrow$ Operator: Backticks `‘...‘`

```
5 ‘mod‘ 2 = 1
```

FUNKTIONSKOMPOSITION

Analog zur mathematischen Notation $f = g \circ h$ für $f(x) = g(h(x))$ versteht auch Haskell das Kompositionsprinzip mit dem Operator `.`
z.B.

```
sqAdd :: Int -> Int
sqAdd = (^2) . (+ 5)
```

statt `sqAdd x = (x + 5)^2` für das Quadrat des fünften Nachfolgers

PARTIELLE APPLIKATION

Funktionen müssen nicht immer mit allen Argumenten versorgt werden. Lässt man (hintere) Argumente weg, so spricht man von Unterversorgung. Die Modulo Funktion hat eigentlich zwei Argumente. Lassen wir das zweite Argument weg, so liefert dies uns eine neue Funktion, die noch ein Argument entgegennimmt und sodann die Restberechnung ausführt.

```
mod :: Int -> Int -> Int
mod m n = ...

mod 10 :: Int -> Int
(mod 10) n = mod 10 n
```

```
(> 3) :: Int -> Bool
(> 3) x = x > 3
```

FUNKTIONEN HÖHERER ORDNUNG — MAP

Funktionen können als Argumente von Funktionen auftreten. Wir lernen drei Basics kennen:

Die Funktion map

- `map` ermöglicht es eine Funktion `f` auf alle Elemente einer Liste anzuwenden

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

- *Beispiel.*

```
map square [1,2,7,12,3,20] = [1,4,49,144,9,400]
```

Die Funktion filter

- `filter p xs` liefert eine Liste, die genau die Elemente von `xs` enthält, welche das Prädikat `p` erfüllen

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x]
```

- *Beispiel.*

```
filter odd [1,2,7,12,3,20] = [1,7,3]
```

Die Funktion foldr

- `foldr f z xs` faltet eine Liste `xs` und verknüpft jeweils durch die Funktion `f`; gestartet wird mit `z` und dem rechtesten Element

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

- *Beispiel.*

```
foldr (+) 3 [1,2,3,4,5] = 18
length xs = foldr (+) 0 (map (\x -> 1) xs)
```

FUNKTIONEN HÖHERER ORDNUNG – ÜBERSICHT

- `map` wendet Funktion auf alle Listenelemente an

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

- `filter` wählt Listenelemente anhand einer Funktion aus

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x]
```

- `foldr` faltet eine Liste mit Verknüpfungsfunktion (von rechts beginnend)

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

Aufgaben 1 & 2

Funktionen höherer Ordnung

AUFGABE 1

Produkt der Quadrate aller geraden Zahlen einer Liste

```haskell
f :: [Int] -> Int
```

```haskell
f xs
  = foldr (+) 0 (map (^2) (filter (`mod` 2) == 0) xs))
```

```haskell
f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

```haskell
f'' = foldr (*) 1 . map (^2) . filter even
```

```haskell
f''' = foldr (*) 1 . map (^2)
                   . filter ((== 0) . (`mod` 2))
```

AUFGABE 2

Faltung einer Liste von *links*

```
foldleft ::  (a -> b -> a) -> a -> [b] -> a
```

```
foldleft ::  (a -> b -> a) -> a -> [b] -> a
foldleft f x []      = x
foldleft f x (y:ys) = foldleft f (f x y) ys
```

Aufgabe 3

Bäume mit beliebig vielen Kindern

Beispielbaum

```
data Tree a = Node a [Tree a] deriving Show
```

```
mytree :: Tree Char
mytree = Node 'a' [
       Node 'b' [ Node 'c' [], Node 'd' [] ] ,
       Node 'e' [ Node 'f' [] ],
       Node 'g' []
      ]
```

Test auf ungerade Anzahl an Kindern

```
data Tree a = Node a [Tree a] deriving Show
oddTree :: Tree a -> Bool
```

```
oddTree :: Tree a -> Bool
oddTree (Node _ []) = True
oddTree (Node _ ts) = (length ts `mod` 2 == 1)
                      && oddTrees ts
  where
    oddTrees :: [Tree a] -> Bool
    oddTrees []       = True
    oddTrees (t : ts) = oddTree t && oddTrees ts
```

Pre-Order-Traversierung

```
data Tree a = Node a [Tree a] deriving Show
preOrder :: Tree a -> [a]
```

```
preOrder :: Tree a -> [a]
preOrder (Node x ts) = x : preOrderTrees ts
  where
    preOrderTrees :: [Tree a] -> [a]
    preOrderTrees []       = []
    preOrderTrees (t : ts) = preOrder t ++
                             preOrderTrees ts
-- alternativ:
preOrder :: Tree a -> [a]
preOrder (Node x ts) = x : concatMap preOrder ts
```

ENDE

Fragen?