

PROGRAMMIERUNG

Übung 2: Haskell – Funktionen höherer Ordnung und Zeichenketten

Eric Kunze

`eric.kunze@mailbox.tu-dresden.de`

TU Dresden, 23. April 2019

Rekapitulation: Funktionsprinzip

Wir erinnern uns an das **Funktionsprinzip**:

Definition durch

- **Basisfall.** 0-äre Wertkonstrukturen
z.B. `[]` oder `Leaf`
- **Rekursionsfall.** (> 0)-äre Wertkonstrukturen
z.B. `:` oder `Node`

Wichtige Hinweise

korrekte Klammerung:

`f (Leaf a) (Leaf b)` statt `f Leaf a Leaf b`

Jeder Ausdruck hat einen Typ:

Statt

`if a == b then boolExp else False`

sollte immer

`a == b && boolExp`

verwendet werden.

Funktionen

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

anonyme Funktionen.

Funktionen ohne konkreten Namen

z.B. $(\lambda x \rightarrow x + 1)$ ist die Addition mit 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.

`prodOdd = prod . filter odd`

statt `prodOdd xs = prod (filter odd xs)` für das Produkt aller ungeraden Listenelemente

Funktionen höherer Ordnung – map

Die Funktion map.

- $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
 $\text{map } f [] = []$
 $\text{map } f (x:xs) = f x : \text{map } f xs$
- `map` ermöglicht es eine Funktion `f` auf alle Elemente einer Liste anzuwenden
- **Beispiel.**
 $\text{map square } [1,2,7,12,3,20] = [1,4,49,144,9,400]$

Funktionen höherer Ordnung – filter

Die Funktion filter.

- `filter :: (a -> Bool) -> [a] -> [a]`
`filter p xs = [x | x <- xs, p x]`
- `filter p xs` liefert eine Liste, die genau die Elemente von `xs` enthält, welche das Prädikat `p` erfüllen
- **Beispiel.**
`filter odd [1,2,7,12,3,20] = [1,7,3]`

Funktionen höherer Ordnung – foldr

Die Funktion foldr.

- $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
 $\text{foldr } f \ z \ [] = z$
 $\text{foldr } f \ z \ (x:xs) = f \ x \ (\text{foldr } f \ z \ xs)$
- $\text{foldr } f \ z \ xs$ faltet eine Liste xs und verknüpft jeweils durch die Funktion f ; gestartet wird mit z und dem rechten Element
- **Beispiel.**
 $\text{foldr } (+) \ 3 \ [1,2,3,4,5] = 18$
 $\text{length } xs = \text{foldr } (+) \ 0 \ (\text{map } (\backslash x \rightarrow 1) \ xs)$
- Analog existiert auch eine Funktion foldl , die eine Liste von links beginnend faltet.

Funktionen höherer Ordnung – Übersicht

- **map** wendet Funktion auf alle Listenelemente an
 $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
 $\text{map } f [] = []$
 $\text{map } f (x:xs) = f x : \text{map } f xs$
- **filter** wählt Listenelemente anhand einer Funktion aus
 $\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$
 $\text{filter } p xs = [x \mid x \leftarrow xs, p x]$
- **foldr** faltet eine Liste mit Verknüpfungsfunktion (von rechts beginnend)
 $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
 $\text{foldr } f z [] = z$
 $\text{foldr } f z (x:xs) = f x (\text{foldr } f z xs)$

Zeichenketten

Characters werden in ' ' eingeschlossen.

Strings werden in ' ' eingeschlossen. → normale double-quotes

Dabei gilt `String = [Char]`.

Überprüfe den Typ in GHCi anhand eigener Beispiele mittels `:t`.

z.B.

```
:t ['a','b'] ==> ['a','b'] :: [Char]
```

```
:t 'a' ==> 'a' :: Char
```

```
:t "ab" ==> "ab" :: [Char]
```