

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 6 vom 20.11.2018: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

16.03.08 2018-12-18

1 [35]



Fahrplan

Teil I: Funktionale Programmierung im Kleinen

- ▶ Einführung
- ▶ Funktionen
- ▶ Algebraische Datentypen
- ▶ Typvariablen und Polymorphie
- ▶ Zyklische Datenstrukturen
- ▶ Funktionen höherer Ordnung I
- ▶ Funktionen höherer Ordnung II

Teil II: Funktionale Programmierung im Großen

Teil III: Funktionale Programmierung im richtigen Leben

PI3 WS 18/19

2 [35]



Inhalt

- ▶ Funktionen **höherer Ordnung**:
 - ▶ Funktionen als gleichberechtigte Objekte
 - ▶ Funktionen als Argumente
- ▶ Spezielle Funktionen: map, filter, fold und Freunde

PI3 WS 18/19

3 [35]



Funktionen als Werte

PI3 WS 18/19

4 [35]



Funktionen Höherer Ordnung

Slogan

"Functions are first-class citizens."

- ▶ Funktionen sind **gleichberechtigt**: Ausdrücke wie **alle anderen**
- ▶ **Grundprinzip** der funktionalen Programmierung
- ▶ Modellierung **allgemeiner Berechnungsmuster**
- ▶ Kontrollabstraktion

PI3 WS 18/19

5 [35]



Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
           | Lager Artikel Menge Lager
```

```
data Eink = ...
           Gelöst durch Polymorphie
           swagen
```

```
data String = Empty
           | Char :+ String
```

- ▶ ein **konstanter** Konstruktor
- ▶ ein **linear rekursiver** Konstruktor

PI3 WS 18/19

6 [35]



Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen -> Int
kasse LeeresWagen = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager -> Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

```
length :: String -> Int
length Empty = 0
length (c :+ s) = 1 + length s
```

Gemeinsamkeiten:

- ▶ ein Fall pro Konstruktor
- ▶ **linearer** rekursiver Aufruf
- ▶ **Nicht** durch Polymorphie gelöst (keine Instanz **einer** Definition)

PI3 WS 18/19

7 [35]



Ein einheitlicher Rahmen

- ▶ Zwei ähnliche Funktionen:

```
toL :: String -> String
toL [] = []
toL (c:cs) = toLower c : toL cs

toU :: String -> String
toU [] = []
toU (c:cs) = toUpper c : toU cs
```

- ▶ Warum nicht **eine** Funktion ... und **zwei** Instanzen?

```
map f [] = []
map f (c:cs) = f c : map f cs
```

```
toL cs = map toLower cs
toU cs = map toUpper cs
```

- ▶ **Funktion f** als **Argument**
- ▶ Was hätte map für einen **Typ**?

PI3 WS 18/19

8 [35]



Funktionen als Werte: Funktionstypen

- Was hätte map für einen **Typ**?

```
map f [] = []  
map f (c:cs) = f c : map f cs
```

- Was ist der Typ des ersten Arguments?
 - Eine Funktion mit beliebigen Definitionsbereich und Wertebereich: $\alpha \rightarrow \beta$
- Was ist der Typ des zweiten Arguments?
 - Eine Liste, auf deren Elemente die Funktion f angewandt wird: $[\alpha]$
- Was ist der Ergebnistyp?
 - Eine Liste von Elementen aus dem Wertebereich von f : $[\beta]$

- Alles **zusammengesetzt**:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```



Map und Filter



Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an

- Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$   
map f [] = []  
map f (c:cs) = f c : map f cs
```

- Auswertung:

```
toL "AB"  $\rightarrow$  map toLower ('A':'B':[])  
 $\rightarrow$  toLower 'A': map toLower ('B':[])  
 $\rightarrow$  'a':map toLower ('B':[])  
 $\rightarrow$  'a':toLower 'B':map toLower []  
 $\rightarrow$  'a':'b':map toLower []  
 $\rightarrow$  'a':'b':[]  $\equiv$  "ab"
```

- Funktionsausdrücke werden symbolisch reduziert

- Keine Änderung



Funktionen als Argumente: filter

- Elemente **filtern**: filter

- Signatur:

```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow [\alpha]$ 
```

- Definition

```
filter p [] = []  
filter p (x:xs)  
  | p x = x : filter p xs  
  | otherwise = filter p xs
```

- Beispiel:

```
letters :: String  $\rightarrow$  String  
letters = filter isAlpha
```



Beispiel filter: Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve (p:ps) = p: sieve (filter ps) where  
  filter (q:qs)  
    | q 'mod' p  $\neq$  0 = q: filter qs  
    | otherwise = filter qs
```

- Einfacher mit filter

- Es wird gefiltert mit $\text{mod } q \ p \neq 0$ (Funktionsparameter q)

- Namenlose** (anonyme) Funktion $\lambda q \rightarrow \text{mod } q \ p \neq 0$

```
sieve :: [Integer]  $\rightarrow$  [Integer]  
sieve (p:ps) = p: sieve (filter ( $\lambda q \rightarrow q \text{ 'mod' } p \neq 0$ ) ps)
```



Funktionen Höherer Ordnung



Funktionen als Argumente: Funktionskomposition

- Funktionskomposition** (mathematisch)

```
( $\circ$ ) :: ( $\beta \rightarrow \gamma$ )  $\rightarrow$  ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $\alpha \rightarrow \gamma$   
( $f \circ g$ ) x = f (g x)
```

- Vordefiniert
- Lies: f nach g

- Funktionskomposition **vorwärts**:

```
(>.>) :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  ( $\beta \rightarrow \gamma$ )  $\rightarrow$   $\alpha \rightarrow \gamma$   
( $f >.> g$ ) x = g (f x)
```

- Nicht** vordefiniert



η -Kontraktion

- " $>.>$ ist dasselbe wie \circ nur mit vertauschten Argumenten"

- Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$   $\beta \rightarrow \alpha \rightarrow \gamma$   
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  ( $\beta \rightarrow \gamma$ )  $\rightarrow$   $\alpha \rightarrow \gamma$   
(>.>) = flip ( $\circ$ )
```

- Da fehlt doch was?!** Nein:

```
(>.>) = flip ( $\circ$ )  $\equiv$  (>.>) f g a = flip ( $\circ$ ) f g a
```

- Warum?



η -Äquivalenz und η -Kontraktion

η -Äquivalenz

Sei f eine Funktion $f : A \rightarrow B$, dann gilt $f = \lambda x. f x$

▶ In Haskell: η -Kontraktion

- ▶ Bedingung: Ausdruck $E :: \alpha \rightarrow \beta$, Variable $x :: \alpha$, E darf x nicht enthalten
 $\lambda x \rightarrow E x \equiv E$

▶ Spezialfall Funktionsdefinition (punktfreie Notation)

$$f x = E x \equiv f = E$$

▶ Hier:

$$(>.>) f g a = \text{flip } (\circ) f g a \equiv (>.>) f g a = \text{flip } (\circ) f g a$$



Partielle Applikation

▶ Funktionskonstruktor rechtsassoziativ:

$$\alpha \rightarrow \beta \rightarrow \gamma \equiv \alpha \rightarrow (\beta \rightarrow \gamma)$$

- ▶ **Inbesondere:** $(\alpha \rightarrow \beta) \rightarrow \gamma \neq \alpha \rightarrow (\beta \rightarrow \gamma)$

▶ Funktionsanwendung ist linksassoziativ:

$$f a b \equiv (f a) b$$

- ▶ **Inbesondere:** $f (a b) \neq (f a) b$

▶ Partielle Anwendung von Funktionen:

- ▶ Für $f :: \alpha \rightarrow \beta \rightarrow \gamma$, $x :: \alpha$ ist $f x :: \beta \rightarrow \gamma$

▶ Beispiele:

- ▶ `map toLower :: String → String`
- ▶ `(3 ==) :: Int → Bool`
- ▶ `concat ◦ map (replicate 2) :: String → String`

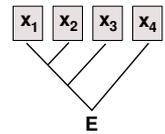


Strukturelle Rekursion

Strukturelle Rekursion

▶ Strukturelle Rekursion: gegeben durch

- ▶ eine Gleichung für die leere Liste
- ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)



- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(+)`, ...

▶ Auswertung:

$$\begin{aligned} \text{sum } [4, 7, 3] &\rightarrow 4 + 7 + 3 + 0 \\ \text{concat } [A, B, C] &\rightarrow A \# B \# C \# [] \\ \text{length } [4, 5, 6] &\rightarrow 1 + 1 + 1 + 0 \end{aligned}$$



Strukturelle Rekursion

▶ Allgemeines Muster:

$$\begin{aligned} f [] &= e \\ f (x:xs) &= x \otimes f xs \end{aligned}$$

▶ Parameter der Definition:

- ▶ Startwert (für die leere Liste) $e :: \beta$
- ▶ Rekursionsfunktion $\otimes :: \alpha \rightarrow \beta \rightarrow \beta$

▶ Auswertung:

$$f [x_1, \dots, x_n] = x_1 \otimes x_2 \otimes \dots \otimes x_n \otimes e$$

- ▶ **Terminiert** immer (wenn Liste endlich und \otimes, e terminieren)



Strukturelle Rekursion durch foldr

▶ Strukturelle Rekursion

- ▶ Basisfall: leere Liste
- ▶ Rekursionsfall: Kombination aus Listenkopf und Rekursionswert

▶ Signatur

$$\text{foldr} :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$

▶ Definition

$$\begin{aligned} \text{foldr } f e [] &= e \\ \text{foldr } f e (x:xs) &= f x (\text{foldr } f e xs) \end{aligned}$$



Beispiele: foldr

▶ **Summieren** von Listenelementen.

$$\begin{aligned} \text{sum} &:: [\text{Int}] \rightarrow \text{Int} \\ \text{sum } xs &= \text{foldr } (+) 0 xs \end{aligned}$$

▶ **Flachklopfen** von Listen.

$$\begin{aligned} \text{concat} &:: [[a]] \rightarrow [a] \\ \text{concat } xs &= \text{foldr } (\#) [] xs \end{aligned}$$

▶ **Länge** einer Liste

$$\begin{aligned} \text{length} &:: [a] \rightarrow \text{Int} \\ \text{length } xs &= \text{foldr } (\lambda x n \rightarrow n + 1) 0 xs \end{aligned}$$



Beispiele: foldr

▶ **Konjunktion** einer Liste

$$\begin{aligned} \text{and} &:: [\text{Bool}] \rightarrow \text{Bool} \\ \text{and } xs &= \text{foldr } (\&\&) \text{True } xs \end{aligned}$$

▶ **Konjunktion** von Prädikaten

$$\begin{aligned} \text{all} &:: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow \text{Bool} \\ \text{all } p &= \text{and } \circ \text{map } p \end{aligned}$$



Der Shoppe, revisited.

► Kasse alt:

```
kasse :: Einkaufswagen → Int
kasse (Ekwg ps) = kasse' ps where
  kasse' [] = 0
  kasse' (p:ps) = cent p + kasse' ps
```

► Kasse neu:

```
kasse' :: Einkaufswagen → Int
kasse' (Ekwg ps) = foldr (\p ps → cent p + ps) 0 ps
```

Besser:

```
kasse :: Einkaufswagen → Int
kasse (Ekwg ps) = sum (map cent ps)
```



Der Shoppe, revisited.

► Inventur alt:

```
inventur :: Lager → Int
inventur (Lager ps) = inventur' ps where
  inventur' [] = 0
  inventur' (p:ps) = cent p + inventur' ps
```

► Suche nach einem Artikel neu:

```
inventur :: Lager → Int
inventur (Lager l) = sum (map cent l)
```



Der Shoppe, revisited.

► Suche nach einem Artikel alt:

```
suche :: Artikel → Lager → Maybe Menge
suche art (Lager ps) = suche' art ps where
  suche' art (Posten lart m: l)
    | art == lart = Just m
    | otherwise = suche' art l
  suche' art [] = Nothing
```

► Suche nach einem Artikel neu:

```
suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) =
  listToMaybe (map (\(Posten _ m) → m)
                (filter (\(Posten la _) → la == a) ps))
```



Der Shoppe, revisited.

► Kassenbon formatieren neu:

```
kassenbon :: Einkaufswagen → String
kassenbon ew@(Ekwg ps) =
  "Bob's Aulde Grocery Shoppe\n" ++
  "ArtikelMengePreis\n" ++
  "-----\n" ++
  concatMap artikel ps ++
  "-----\n" ++
  "Summe:" ++ formatR 31 (showEuro (kasse ew))
```

```
artikel :: Posten → String
```



Noch ein Beispiel: rev

► Listen umdrehen:

```
rev1 :: [a] → [a]
rev1 [] = []
rev1 (x:xs) = rev1 xs ++ [x]
```

► Mit foldr:

```
rev2 :: [a] → [a]
rev2 = foldr (\x xs → xs ++ [x]) []
```

► Unbefriedigend: doppelte Rekursion $O(n^2)$!



Iteration mit foldl

► foldr faltet von rechts:

$\text{foldr } \otimes [x_1, \dots, x_n] e = x_1 \otimes x_2 (x_2 \otimes (\dots (x_n \otimes e)))$

► Warum nicht andersherum?

$\text{foldl } \otimes [x_1, \dots, x_n] e = (((e \otimes x_1) \otimes x_2) \dots) \otimes x_n$

► Definition von foldl:

```
foldl :: (α → β → α) → α → [β] → α
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

► foldl ist ein **Iterator** mit Anfangszustand e, Iterationsfunktion \otimes

► Entspricht einfacher Iteration (for-Schleife)



foldr vs. foldl

► $f = \text{foldr } \otimes e$ entspricht

```
f [] = e
f (x:xs) = x \otimes f xs
```

- **Nicht-strikt** in xs, z.B. and, or
- Konsumiert nicht immer die ganze Liste
- Auch für unendliche Listen anwendbar

► $f = \text{foldl } \otimes e$ entspricht

```
f xs = g e xs where
  g a [] = a
  g a (x:xs) = g (a \otimes x) xs
```

- **Effizient** (endrekursiv) und **strikt** in xs
- Konsumiert immer die ganze Liste
- Divergiert immer für unendliche Listen



Beispiel: rev revisited

► Listenumkehr **endrekursiv**:

```
rev3 :: [a] → [a]
rev3 xs = rev0 xs [] where
  rev0 [] ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

► Listenumkehr durch falten **von links**:

```
rev4 :: [a] → [a]
rev4 = foldl (\xs x → x:xs) []
```

```
rev5 :: [a] → [a]
rev5 = foldl (flip (:)) []
```

► Nur noch **eine** Rekursion $O(n)$!



Wann ist foldl = foldr?

Definition (Monoid)

(\otimes, A) ist ein **Monoid** wenn

$$\begin{aligned} A \otimes x &= x && \text{(Neutrales Element links)} \\ x \otimes A &= x && \text{(Neutrales Element rechts)} \\ (x \otimes y) \otimes z &= x \otimes (y \otimes z) && \text{(Assoziativität)} \end{aligned}$$

Theorem

Wenn (\otimes, A) **Monoid**, dann für alle A, xs

$$\text{foldl } \otimes A xs = \text{foldr } \otimes A xs$$

- ▶ Beispiele: length, concat, sum
- ▶ Gegenbeispiele: rev, all



Übersicht: vordefinierte Funktionen auf Listen II

```
map      :: (α → β) → [α] → [β]      — Auf alle anwenden
filter  :: (α → Bool) → [α] → [α]     — Elemente filtern
foldr   :: (α → β → β) → β → [α] → β — Falten von rechts
foldl   :: (β → α → β) → β → [α] → β — Falten von links
mapConcat :: (α → [β]) → [α] → [β]    — map und concat
takeWhile :: (α → Bool) → [α] → [α]   — längster Prefix mit p
dropWhile :: (α → Bool) → [α] → [α]   — Rest von takeWhile
span    :: (α → Bool) → [α] → ([α], [α]) — takeWhile und dropWhile
all     :: (α → Bool) → [α] → Bool     — p gilt für alle
any     :: (α → Bool) → [α] → Bool     — p gilt mind. einmal
elem    :: (Eq α) ⇒ α → [α] → Bool    — Ist Element enthalten?
zipWith :: (α → β → γ) → [α] → [β] → [γ] — verallgemeinertes zip
```

- ▶ Mehr: siehe Data.List



Zusammenfassung

- ▶ Funktionen **höherer Ordnung**
 - ▶ Funktionen als gleichberechtigte Objekte und Argumente
 - ▶ Partielle Applikation, η-Kontraktion, namenlose Funktionen
 - ▶ Spezielle Funktionen höherer Ordnung: map, filter, fold und Freunde
- ▶ Formen der **Rekursion**:
 - ▶ Strukturelle Rekursion entspricht foldr
 - ▶ Iteration entspricht foldl
- ▶ Nächste Woche: fold für andere Datentypen

