Praktische Informatik 3: Funktionale Programmierung

Vorlesung 5 vom 30.11.2020: Funktionen Höherer Ordnung I

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2020/21

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Rekursive und zyklische Datenstrukturen
  - Funktionen höherer Ordnung II
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

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

Lernziel

Wir verstehen, wie wir mit `map`, `filter` und `fold` wiederkehrende Funktionsmuster kürzer und verständlicher aufschreiben können, und wir verstehen, warum der Funktionstyp in $\alpha \to \beta$ ein Typ wie jeder andere ist.

I. Funktionen als Werte

Funktionen Höherer Ordnung

Slogan

"Functions are first-class citizens."

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

Ähnliche Datentypen der letzten Vorlesung

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

```
data Einkaufskorb = LeererKorb
                  | Einkauf Artikel Menge Einkaufskorb
```

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

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

Ähnliche Datentypen der letzten Vorlesung

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

```haskell
data Einkaufskorb = LeererKorb
                  | Einkauf Artikel Menge Einkaufskorb
```

```haskell
data MyString = Empty
              | Char :+ MyString
```

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

Gelöst durch Polymorphie

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb→ Int
kasse LeererKorb = 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 :: MyString→ Int
length Empty     = 0
length (c :+ s) = 1+ length s
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- **linearer** rekursiver Aufruf

Ähnliche Funktionen der letzten Vorlesung

```haskell
kasse :: Einkaufskorb→ Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m+ kasse e
```

```haskell
inventur :: Lager→ Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m+ inventur l
```

```haskell
length :: MyString→ Int
length Empty     = 0
length (c :+ s) = 1+ length s
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- **linearer** rekursiver Aufruf

Nicht durch Polymorphie gelöst

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 . . .

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 ...

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

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**?

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?

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 Definitions- und Wertebereich: $\alpha \rightarrow \beta$
- Was ist der Typ des zweiten Arguments?

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 Definitions- und Wertebereich: $\alpha \to \beta$
- Was ist der Typ des zweiten Arguments?
  - Eine Liste, auf deren Elemente die Funktion f angewant wird: $[\alpha]$
- Was ist der Ergebnistyp?

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 Definitions- und Wertebereich: $\alpha \rightarrow \beta$
- Was ist der Typ des zweiten Arguments?
  - Eine Liste, auf deren Elemente die Funktion f angewant wird: $[\alpha]$
- Was ist der Ergebnistyp?
  - Eine Liste von Elementen aus dem Wertebereich von `f`: $[\beta]$
- Alles **zusammengesetzt**:

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 Definitions- und Wertebereich: $\alpha \to \beta$
- Was ist der Typ des zweiten Arguments?
  - Eine Liste, auf deren Elemente die Funktion f angewant wird: $[\alpha]$
- Was ist der Ergebnistyp?
  - Eine Liste von Elementen aus dem Wertebereich von `f`: $[\beta]$
- Alles **zusammengesetzt**:

```
map :: (α → β) → [α] → [β]
```

Was zum Selberdenken.

Die **konstante** Funktion ist

```
const :: α → β → α
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Was zum Selberdenken.

Die **konstante** Funktion ist

```
const :: α → β → α
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Lösung: Betrachten wir eine Beispiel-Auswertung:

```
sum (map (const 1) [])  ⇝  sum []  ⇝  0
sum (map (const 1) [True, False, True])  ⇝  sum [1,1,1]  ⇝  3
sum (map (const 1) "foobaz")  ⇝  sum ([1,1,1,1,1,1])  ⇝  6
```

Was zum Selberdenken.

Die **konstante** Funktion ist

```
const :: α→ β→ α
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Lösung: Betrachten wir eine Beispiel-Auswertung:

```
sum (map (const 1) [])  ⇝  sum []  ⇝  0
sum (map (const 1) [True, False, True])  ⇝  sum [1,1,1]  ⇝  3
sum (map (const 1) "foobaz")  ⇝  sum ([1,1,1,1,1,1])  ⇝  6
```

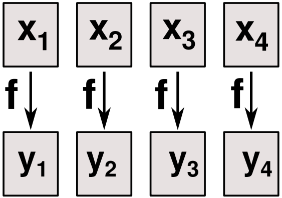
Die mysteriöse Funktion berechnet die **Länge** der Liste `xs`!

II. Map und Filter

Funktionen als Argumente: `map`

- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```

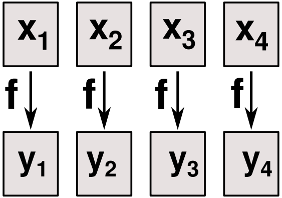


- Auswertung:
  `toL "AB"`

Funktionen als Argumente: `map`

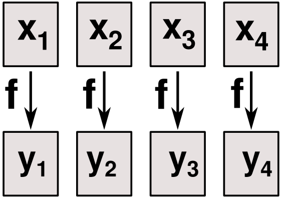
- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```



- Auswertung:
  `toL "AB"` → `map toLower ('A':'B':[])`

Funktionen als Argumente: `map`



- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```

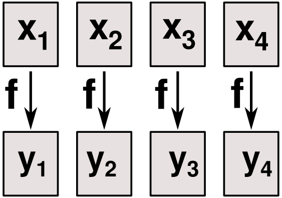
- Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
          → toLower 'A': map toLower ('B':[])
```

Funktionen als Argumente: `map`

- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```



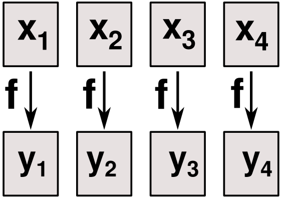
- Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
          → toLower 'A': map toLower ('B':[])
          → 'a':map toLower ('B':[])
```

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α → β) → [α] → [β]
map f []     = []
map f (c:cs) = f c : map f cs
```



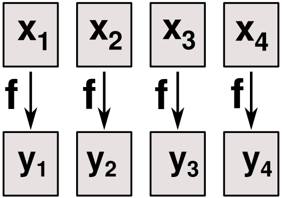
- Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
          → toLower 'A': map toLower ('B':[])
          → 'a':map toLower ('B':[])
          → 'a':toLower 'B':map toLower []
```

Funktionen als Argumente: `map`

- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```



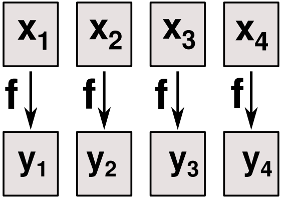
- Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
          → toLower 'A': map toLower ('B':[])
          → 'a':map toLower ('B':[])
          → 'a':toLower 'B':map toLower []
          → 'a':'b':map toLower []
```

Funktionen als Argumente: map

- `map` wendet Funktion auf alle Elemente an
- Signatur:

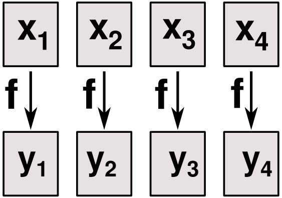
```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```



- Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
          → toLower 'A': map toLower ('B':[])
          → 'a':map toLower ('B':[])
          → 'a':toLower 'B':map toLower []
          → 'a':'b':map toLower []
          → 'a':'b':[]
```

Funktionen als Argumente: `map`



- `map` wendet Funktion auf alle Elemente an
- Signatur:

```
map  ::  (α→ β)→  [α]→  [β]
map f []        = []
map f (c:cs) = f c : map f cs
```

- Auswertung:

```
toL "AB"   → map toLower ('A':'B':[])
           → toLower 'A': map toLower ('B':[])
           → 'a':map toLower ('B':[])
           → 'a':toLower 'B':map toLower []
           → 'a':'b':map toLower []
           → 'a':'b':[] ≡ "ab"
```

- **Funktionsausdrücke** werden symbolisch reduziert
  - Keine Änderung

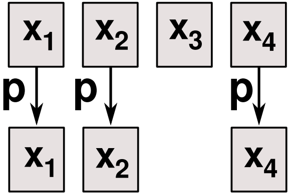
Funktionen als Argumente: filter

- Elemente **filtern**: `filter`
- Signatur:

```
filter :: (α → Bool) → [α] → [α]
```

- Definition

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



- Beispiel:

```
digits :: String → String
digits = filter isDigit
```

Beispiel `filter`: Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve' :: [Integer] → [Integer]
sieve' (p:ps) = p: sieve' (filterPs ps) where
   filterPs (q: qs)
     | q `mod` p ≠ 0 = q: filterPs qs
     | otherwise     = filterPs qs
```

- „Sieb“: es werden alle `q` gefiltert mit `mod q p` $\neq$ `0`

Beispiel `filter`: Sieb des Erathostenes

- Es werden alle `q` gefiltert mit `mod q p` $\neq$ `0`
- **Namenlose** (anonyme) Funktion λ`q`$\rightarrow$ `mod q p` $\neq$ `0`

```
sieve :: [Integer] → [Integer]
sieve (p:ps) = p: sieve (filter (λq→ q `mod` p ≠ 0) ps)
```

- Damit (NB: kleinste Primzahl ist `2`):

```
primes :: [Integer]
primes = sieve [2..]
```

Beispiel `filter`: Sieb des Erathostenes

- Es werden alle `q` gefiltert mit `mod q p` $\neq$ `0`
- **Namenlose** (anonyme) Funktion λ`q`$\rightarrow$ `mod q p` $\neq$ `0`

```
sieve :: [Integer] → [Integer]
sieve (p:ps) = p: sieve (filter (λq → q `mod` p ≠ 0) ps)
```

- Damit (NB: kleinste Primzahl ist `2`):

```
primes :: [Integer]
primes = sieve [2..]
```

- Primzahlzählfunktion $\pi(n)$:

```
pcf :: Integer → Int
pcf n = length (takeWhile (λm → m < n) primes)
```

Primzahltheorem:

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\log n} = 1$$

Jetzt seit ihr dran. . .

Für das Palindrom hatten wir eine Funktion `clean` definiert:

```
clean :: String→ String
clean [] = []
clean (s:xs) | isAlphaNum s = toLower s : clean xs
             | otherwise    = clean xs
```

Übung 5.2: `Clean` refactored

Wie sieht `clean` mit `map` und `filter` (und **ohne Rekursion**) aus?

Jetzt seit ihr dran. . .

Für das Palindrom hatten wir eine Funktion `clean` definiert:

```
clean :: String→ String
clean [] = []
clean (s:xs) | isAlphaNum s = toLower s : clean xs
             | otherwise    = clean xs
```

Übung 5.2: `Clean` refactored

Wie sieht `clean` mit `map` und `filter` (und **ohne Rekursion**) aus?

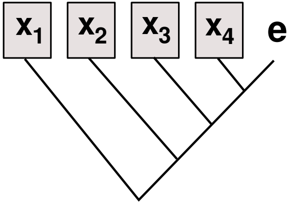
Lösung:

```
clean' :: String→ String
clean' xs = map toLower (filter isAlphaNum xs)
```

III. 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)



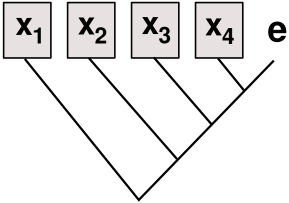
- Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...

- Auswertung:

```
sum [4,7,3]        →
concat [A, B, C]   →
length [4, 5, 6]   →
```

Strukturelle Rekursion

- **Strukturelle Rekursion**: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)

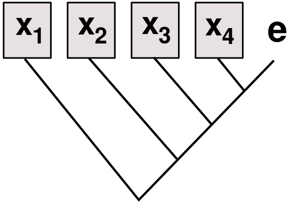


- Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- Auswertung:

```
sum [4,7,3]      →  4 + 7 + 3 + 0
concat [A, B, C] →
length [4, 5, 6] →
```

Strukturelle Rekursion

- **Strukturelle Rekursion**: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)

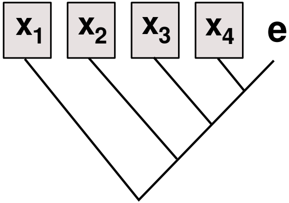


- Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- Auswertung:

```
sum [4,7,3]          →  4 + 7 + 3 + 0
concat [A, B, C]     →  A ++ B ++ C++ []
length [4, 5, 6]     →
```

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:

```
sum [4,7,3]        →   4 + 7 + 3 + 0
concat [A, B, C]   →   A ++ B ++ C++ []
length [4, 5, 6]   →   1+ 1+ 1+ 0
```

Strukturelle Rekursion

- **Allgemeines Muster**:

```
f []     = e
f (x:xs) = x ⊗ f xs
```

- Parameter der Definition:
  - Startwert (für die leere Liste) e :: β
  - Rekursionsfunktion $\otimes$ :: $\alpha \to \beta \to \beta$
- Auswertung:

$$\texttt{f [x1,..., xn]} = \texttt{x1} \otimes \texttt{x2} \otimes \ldots \otimes \texttt{xn} \otimes \texttt{e}$$

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

Strukturelle Rekursion durch `foldr`

- **Stukturelle** Rekursion
  - Basisfall: leere Liste
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- Signatur

```
foldr :: (α→ β→ β)→ β→ [α]→ β
```

- Definition

```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Beispiele: foldr

- **Summieren** von Listenelementen.

```
sum :: [Int]→ Int
sum xs = foldr (+) 0 xs
```

- **Flachklopfen** von Listen.

```
concat :: [[a]]→ [a]
concat xs = foldr (++) [] xs
```

- **Länge** einer Liste

```
length :: [a]→ Int
length xs = foldr (λx n→ n+ 1) 0 xs
```

Beispiele: `foldr`

- **Konjunktion** einer Liste

```
and :: [Bool] → Bool
and xs = foldr (&&) True xs
```

- **Konjunktion** von Prädikaten

```
all :: (α→ Bool)→ [α] → Bool
all p xs = and (map p xs)
```

Der Shoppe, revisited.

- Kasse alt:

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

- Kasse neu:

```
kasse' :: Einkaufskorb→ Int
kasse' (Ek ps) = foldr (λp ps→ cent p+ ps) 0 ps
```

  Besser:

```
kasse :: Einkaufskorb→ Int
kasse (Ek 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 (λ(Posten _ m)→ m)
                (filter (λ(Posten la _) → la == a) ps))
```

Der Shoppe, revisited.

- Kassenbon formatieren neu:

```
kassenbon :: Einkaufskorb-> String
kassenbon ek@(Ek ps) =
  "Bob's␣Aulde␣Grocery␣Shoppe\n\n"++
  "Artikel␣␣␣␣␣␣␣␣␣␣␣␣␣␣Menge␣␣␣␣␣␣Preis\n"++
  "-------------------------------------\n"++
  concatMap artikel ps ++
  "=====================================\n"++
  "Summe:"++ formatR 31 (showEuro (kasse ek))
```

```
artikel :: Posten-> String
```

Noch ein Beispiel: `rev`

- Listen **umdrehen**:

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

- Mit `foldr`:

```
rev2 :: [α]→ [α]
rev2 = foldr (λx xs→ xs++ [x]) []
```

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

Iteration mit `foldl`

- `foldr` faltet von rechts:

$$\texttt{foldr} \otimes \texttt{[x1,.., xn] e} = \texttt{x1} \otimes \texttt{x2 (x2} \otimes \texttt{(...(xn} \otimes \texttt{e)))}$$

Iteration mit `foldl`

- `foldr` faltet von rechts:

  $$\texttt{foldr} \otimes \texttt{[x1,..., xn] e} = \texttt{x1} \otimes \texttt{x2 (x2} \otimes \texttt{(...(xn} \otimes \texttt{e)))}$$

- Warum nicht **andersherum**?

  $$\texttt{foldl} \otimes \texttt{[x1,..., xn] e} = \texttt{(((e} \otimes \texttt{x1)} \otimes \texttt{x2)..)} \otimes \texttt{xn}$$

Iteration mit `foldl`

- `foldr` faltet von rechts:

  $$\texttt{foldr} \otimes \texttt{[x1,.., xn] e} = \texttt{x1} \otimes \texttt{x2 (x2} \otimes \texttt{(...(xn} \otimes \texttt{e)))}$$

- Warum nicht **andersherum**?

  $$\texttt{foldl} \otimes \texttt{[x1,.., xn] e} = \texttt{(((e} \otimes \texttt{x1)} \otimes \texttt{x2).)} \otimes \texttt{xn}$$

- 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)

Beispiel: `rev` revisited

- Listenumkehr **endrekursiv**:

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

- Listenumkehr durch falten **von links**:

```
rev4 :: [α]→ [α]
rev4 = foldl (λxs x → x: xs) []
```

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

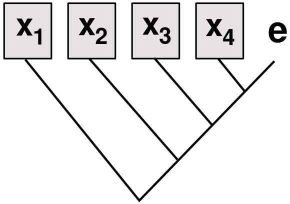
- Nur noch **eine** Rekursion $O(n)$!

foldr vs. foldl

- f = foldr ⊗ e entspricht

```
f []     = e
f (x:xs) = x ⊗ f xs
```

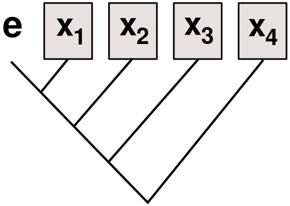
  - **Nicht-strikt** in xs, z.B. and, or
  - Konsumiert nicht immer die ganze Liste
  - Auch für unendliche Listen anwendbar



- f = foldl ⊗ e entspricht

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

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



Wann ist `foldl` = `foldr`?

Definition (Monoid)

$(\otimes, e)$ ist ein **Monoid** wenn

$$e \otimes x = x \qquad \text{(Neutrales Element links)}$$

$$x \otimes e = x \qquad \text{(Neutrales Element rechts)}$$

$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \qquad \text{(Assoziativät)}$$

Theorem

*Wenn $(\otimes, e)$ **Monoid** und $\otimes$ strikt, dann gilt für alle e, xs*

$$foldl\ \otimes\ e\ xs = foldr\ \otimes\ e\ xs$$

- Beispiele: `concat`, `sum`, `product`, `length`, `reverse`
- Gegenbeispiel: `all`, `any` (nicht-strikt)

Übersicht: vordefinierte Funktionen auf Listen II

```
map       :: (α→ β)→ [α]→ [β]                 -- Auf alle Elemente 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               -- Argument gilt für alle
any  :: (α → Bool) → [α] → Bool               -- Argument 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`

Jetzt seit ihr dran!

Übung 5.3: `elem` selbstgemacht

Wie könnte die vordefinierte Funktion

$$\texttt{elem} :: (\texttt{Eq}\ \alpha) \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \texttt{Bool}$$

definiert sein?

Jetzt seit ihr dran!

Übung 5.3: `elem` selbstgemacht

Wie könnte die vordefinierte Funktion

$\texttt{elem} :: (\texttt{Eq}\ \alpha) \Rightarrow \alpha \to [\alpha] \to \texttt{Bool}$

definiert sein?

Lösung: Eine Möglichkeit:

$\texttt{elem x xs} = \texttt{not (null (filter (}\lambda\texttt{y}\to\texttt{ x} == \texttt{y) xs))}$

Jetzt seit ihr dran!

Übung 5.3: `elem` selbstgemacht

Wie könnte die vordefinierte Funktion

$\texttt{elem} :: (\texttt{Eq}\ \alpha) \Rightarrow \alpha \to [\alpha] \to \texttt{Bool}$

definiert sein?

Lösung: Eine Möglichkeit:

$\texttt{elem x xs} = \texttt{not (null (filter (}\lambda\texttt{y}\to\texttt{ x} == \texttt{y) xs))}$

oder auch

$\texttt{elem x} = \texttt{not} \circ \texttt{null} \circ \texttt{filter (x} ==\texttt{)}$

IV. Funktionen Höherer Ordnung

Funktionen als Argumente: Funktionskomposition

- **Funktionskomposition** (mathematisch)

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

  - Vordefiniert
  - Lies: `f` nach `g`

- Funktionskomposition **vorwärts**:

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

  - **Nicht** vordefiniert

η-Kontraktion

- "$>.>$ ist dasselbe wie $\circ$ nur mit vertauschten Argumenten"
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

η-Kontraktion

- "$>.>$ ist dasselbe wie $\circ$ nur mit vertauschten Argumenten"
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- **Da fehlt doch was?!**

η-Kontraktion

- "$>.>$ ist dasselbe wie $\circ$ nur mit vertauschten Argumenten"
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- **Da fehlt doch was?!** Nein:

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

- Warum?

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-**Kontraktion**
  - Bedingung: Ausdruck `E` $::\ \alpha \to \beta$, Variable `x` $::\ \alpha$, `E` darf `x` nicht enthalten

$$\lambda \mathtt{x} \to \mathtt{E\ x} \quad \equiv \quad \mathtt{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathtt{f\ x} = \mathtt{E\ x} \quad \equiv \quad \mathtt{f} = \mathtt{E}$$

- Hier:
  `(>.>) f g a = flip (∘) f g a` $\equiv$ `(>.>) f g a = flip (∘) f g a`

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-**Kontraktion**
  - Bedingung: Ausdruck `E` $::$ $\alpha \to \beta$, Variable `x` $::$ α, `E` darf `x` nicht enthalten

$$\lambda\mathtt{x} \to \mathtt{E\ x} \quad \equiv \quad \mathtt{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathtt{f\ x} = \mathtt{E\ x} \quad \equiv \quad \mathtt{f} = \mathtt{E}$$

- Hier:

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

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-**Kontraktion**
  - Bedingung: Ausdruck `E` $::\ \alpha \to \beta$, Variable `x` $::\ \alpha$, `E` darf `x` nicht enthalten

$$\lambda\mathtt{x} \to\ \mathtt{E\ x} \quad \equiv \quad \mathtt{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathtt{f\ x} = \mathtt{E\ x} \quad \equiv \quad \mathtt{f} = \mathtt{E}$$

- Hier:

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

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-**Kontraktion**
  - Bedingung: Ausdruck `E` :: $\alpha \to \beta$, Variable `x` :: α, `E` darf `x` nicht enthalten

$$\lambda\mathtt{x} \to \mathtt{E\ x} \quad \equiv \quad \mathtt{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathtt{f\ x} = \mathtt{E\ x} \quad \equiv \quad \mathtt{f} = \mathtt{E}$$

- Hier:

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

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

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

- Funktionsanwendung ist linksassoziativ:

$$\texttt{f a b} \equiv \texttt{(f a) b}$$

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

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

  - **Inbesondere**: $(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$
- Funktionsanwendung ist linksassoziativ:

$$\texttt{f a b} \equiv \texttt{(f a) b}$$

  - **Inbesondere**: `f (a b)` $\neq$ `(f a) b`
- **Partielle** Anwendung von Funktionen:
  - Für `f` $:: \alpha \to \beta \to \gamma$, `x` $:: \alpha$ ist `f x` $:: \beta \to \gamma$
- Beispiele:
  - `map toLower` :: `String` $\to$ `String`
  - `(3 ==)` :: `Int` $\to$ `Bool`
  - `concat` $\circ$ `map (replicate 2)` :: `String` $\to$ `String`

Zusammenfassung

- Funktionen **höherer Ordnung**
  - Funktionen als gleichberechtigte Objekte und Argumente
  - Spezielle Funktionen höherer Ordnung: `map`, `filter`, `fold` und Freunde
- Formen der **Rekursion**:
  - Strukturelle Rekursion entspricht `foldr`
  - Iteration entspricht `foldl`
- Partielle Applikation, η-Äquivalenz, namenlose Funktionen
- Nächste Woche: Rekursive und zyklische Datenstrukturen