Praktische Informatik 3: Funktionale Programmierung
Vorlesung 5 vom 11.11.2014: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- 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

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path→ Path→ Path
cat Mt q          = q
cat (Cons i p) q = Cons i (cat p q)
```

```
rev :: Path→ Path
rev Mt          = Mt
rev (Cons i p) = cat (rev p) (Cons i Mt)
```

- Zeichenketten:

```
cat :: MyString→ MyString→ MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

```
rev :: MyString→ MyString
rev Empty       = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path→ Path→ Path
cat Mt q          = q
cat (Cons i p) q = Cons i (cat p q)
```

```
rev :: Path→ Path
rev Mt
rev (Co
```

Gelöst durch Polymorphie

- Zeichenl

```
cat :: MyString→ MyString→ MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

```
rev :: MyString→ MyString
rev Empty       = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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
```

```
len :: MyString→ Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- linearer rekursiver Aufruf

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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
```

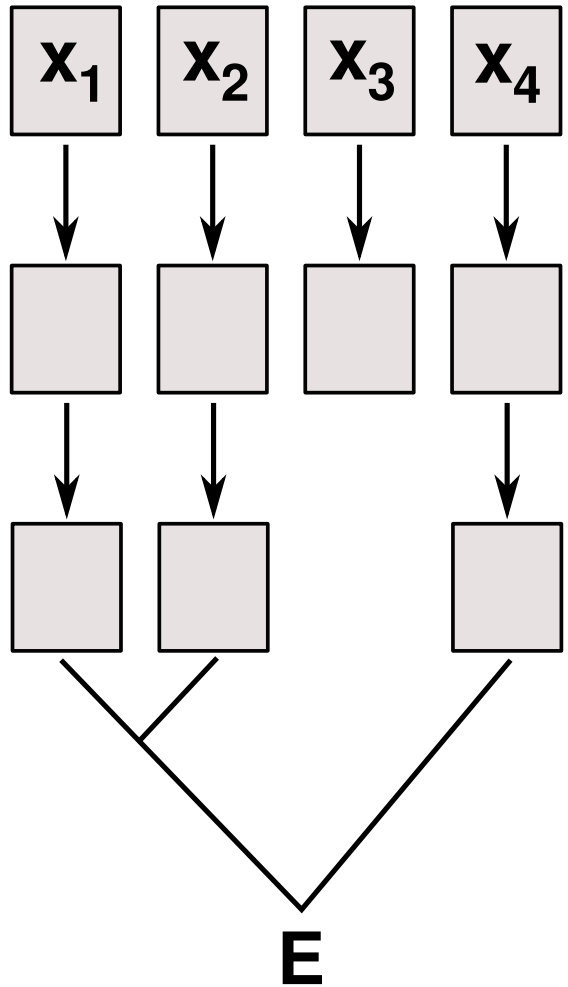
```
len :: MyString→ Int
len Empty        = 0
len (Cons c str) = 1+ len str
```

Gemeinsamkeiten:

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

Muster der primitiven Rekursion

- Anwenden einer Funktion auf jedes Element der Liste
- möglicherweise Filtern bestimmter Elemente
- Kombination der Ergebnisse zu einem Gesamtergebnis E



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 : toL cs
```

- Warum nicht ...

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 : toL cs
```

- Warum nicht ...

```
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 Höherer Ordnung

Slogan

"Functions are first-class citizens."

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

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- Auswertung:
  toL "AB"

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

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

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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"

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:

```
letters :: String→ String
letters = filter isAlpha
```

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: filter (\n→ mod n p ≠0) ps
  - Namenlose (anonyme) Funktion

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: filter (\n→ mod n p ≠0) ps
  - Namenlose (anonyme) Funktion

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

- Alle Primzahlen:

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

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: filter (\n→ mod n p ≠0) ps
  - Namenlose (anonyme) Funktion

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

- Alle Primzahlen:

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

- Die ersten n Primzahlen:

```
n_primes :: Int→ [Integer]
n_primes n = take n primes
```

Funktionen als Argumente: Funktionskomposition

- Funktionskomposition (mathematisch)

```
(∘) :: (β→ γ) → (α→ β)→ α→ γ
(f ∘ g) x = f (g x)
```

  - Vordefiniert
  - Lies: f nach g

- Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(f >.> g) x = g (f x)
```

  - Nicht vordefiniert!

η-Kontraktion

- Vertauschen der Argumente (vordefiniert):

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

η-Kontraktion

- 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

- 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:

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

- Warum?

η-Äquivalenz und *eta*-Kontraktion

η-Äquivalenz

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

- Warum? Extensionale Gleichheit von Funktionen
- In Haskell: η-Kontraktion
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

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

- Syntaktischer Spezialfall Funktionsdefinition (punktfreie Notation)

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$a \to b \to c \equiv a \to (b \to c)$$

  - Inbesondere: $(a \to b) \to c \neq a \to (b \to c)$

- Funktionsanwendung ist linksassoziativ:

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

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$a \to b \to c \equiv a \to (b \to c)$$

  - Inbesondere: $(a \to b) \to c \neq a \to (b \to c)$

- Funktionsanwendung ist linksassoziativ:

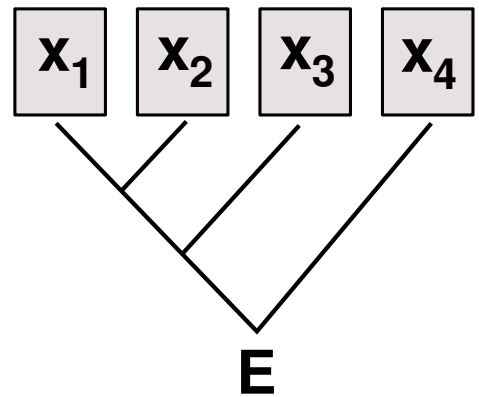
$$f\ a\ b \equiv (f\ a)\ b$$

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

- Partielle Anwendung von Funktionen:
  - Für $f :: a \to b \to c$, $x :: a$ ist $f\ x :: b \to c$ (closure)
- Beispiele:
  - $\text{map toLower} :: \text{String} \to \text{String}$
  - $(3 ==) :: \text{Int} \to \text{Bool}$
  - $\text{concat} \circ \text{map}\ (\text{replicate}\ 2) :: \text{String} \to \text{String}$

Einfache Rekursion

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



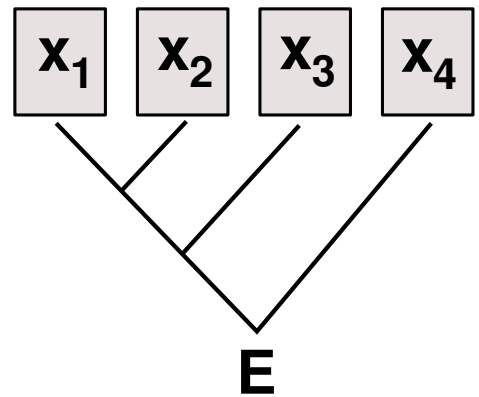
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

sum [4,7,3] $\rightarrow$
concat [A, B, C] $\rightarrow$
length [4, 5, 6] $\rightarrow$

Einfache Rekursion

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



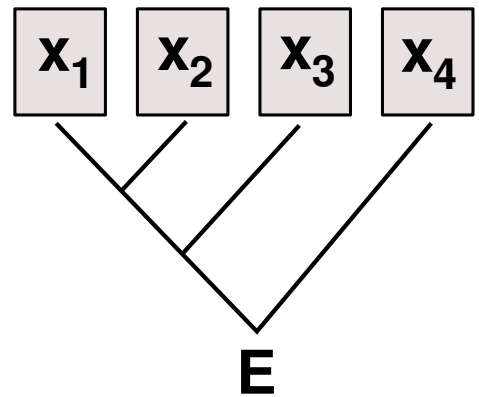
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

sum [4,7,3]	$\rightarrow$	$4 + 7 + 3 + 0$
concat [A, B, C]	$\rightarrow$	
length [4, 5, 6]	$\rightarrow$	

Einfache Rekursion

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



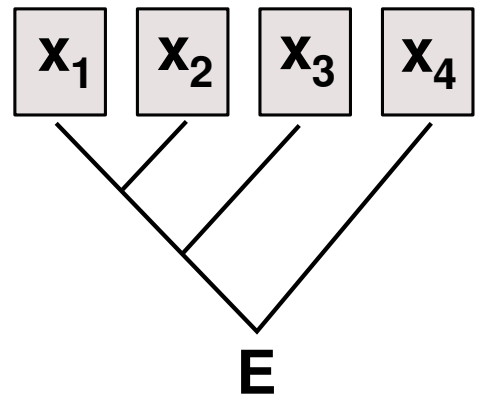
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

sum [4,7,3]	$\rightarrow$	$4 + 7 + 3 + 0$
concat [A, B, C]	$\rightarrow$	$A +\!\!+ B +\!\!+ C +\!\!+ [\,]$
length [4, 5, 6]	$\rightarrow$	

Einfache Rekursion

- Einfache 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:

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

Einfache Rekursion

- Allgemeines Muster:

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

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

$$\mathsf{f}\ [\mathsf{x1},\ldots,\ \mathsf{xn}] = \mathsf{x1} \otimes \mathsf{x2} \otimes \ldots \otimes \mathsf{xn} \otimes \mathsf{A}$$

- Terminiert immer (wenn Liste endlich und $\otimes, A$ terminieren)
- Entspricht einfacher Iteration (while-Schleife)

Einfach Rekursion durch foldr

- Einfache 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 = and ∘ map p
```

Der Shoppe, revisited.

- Suche nach einem Artikel alt:

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

- Suche nach einem Artikel neu:

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

Der Shoppe, revisited.

- Kasse alt:

```
kasse :: Einkaufswagen→ Int
kasse (Einkaufswagen []) = 0
kasse (Einkaufswagen (p: e)) = cent p+ kasse (Einkaufswagen e)
```

- Kasse neu:

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

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

Der Shoppe, revisited.

- Kassenbon formatieren neu:

```
kassenbon :: Einkaufswagen→ String
kassenbon ew@(Einkaufswagen as) =
  "Bob's␣Aulde␣Grocery␣Shoppe\n\n"++
  "Artikel␣␣␣␣␣␣␣␣␣␣␣␣␣␣Menge␣␣␣␣␣␣Preis\n"++
  "──────────────────────────────────────\n"++
  concatMap artikel as ++
  "══════════════════════════════════════\n"++
  "Summe:"++ formatR 31 (showEuro (kasse ew))
```

```
artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"
```

Noch ein Beispiel: rev

- Listen umdrehen:

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

- Mit fold:

```
rev' = foldr snoc []
```

```
snoc :: α → [α] → [α]
snoc x xs = xs ++ [x]
```

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

Einfache Rekursion durch foldl

- foldr faltet von rechts:

$$\text{foldr} \otimes [x_1, ..., x_n]\ A = x_1 \otimes (x_2 \otimes (\ldots (x_n \otimes A)))$$

- Warum nicht andersherum?

$$\text{foldl} \otimes [x_1, ..., x_n]\ A = (((A \otimes x_1) \otimes x_2) \ldots) \otimes x_n$$

- 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
```

Beispiel: rev revisited

- Listenumkehr ist falten von links:

```
rev' xs = foldl (flip (:)) [] xs
```

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

foldr vs. foldl

- f = foldr $\otimes$ A entspricht

$$\begin{array}{lcl} \mathsf{f}\ [] & = & \mathsf{A} \\ \mathsf{f}\ (\mathsf{x}{:}\mathsf{xs}) & = & \mathsf{x} \otimes \mathsf{f}\ \mathsf{xs} \end{array}$$

  - Kann nicht-strikt in xs sein, z.B. and, or
  - Konsumiert nicht immer die ganze Liste
  - Auch für nichtendliche Listen anwendbar

- f = foldl $\otimes$ A entspricht

$$\begin{array}{lcl} \mathsf{f}\ \mathsf{xs} & = & \mathsf{g}\ \mathsf{A}\ \mathsf{xs} \\ \mathsf{g}\ \mathsf{a}\ [] & = & \mathsf{a} \\ \mathsf{g}\ \mathsf{a}\ (\mathsf{x}{:}\mathsf{xs}) & = & \mathsf{g}\ (\mathsf{a} \otimes \mathsf{x})\ \mathsf{xs} \end{array}$$

  - Endrekursiv (effizient) und strikt in xs
  - Konsumiert immer die ganze Liste
  - Divergiert immer für nichtendliche Listen

foldl = foldr

Definition (Monoid)

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

$$A \otimes x = x \qquad \text{(Neutrales Element links)}$$
$$x \otimes A = x \qquad \text{(Neutrales Element rechts)}$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \qquad \text{(Assoziativät)}$$

Theorem

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

$$foldl \otimes A\ xs = foldr \otimes A\ xs$$

- 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 v. rechts
foldl     :: (β→ α→ β)→ β→ [α]→ β          — Falten v. 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])    — take und drop
any       :: (α → Bool) → [α] → Bool       — p gilt mind. einmal
all       :: (α → Bool) → [α] → Bool       — p gilt für alle
elem      :: (Eq α) ⇒ α → [α] → Bool       — Ist enthalten?
zipWith   :: (α → β → γ) → [α] → [β] → [γ]
                                           — verallgemeinertes zip
```

Funktionen Höherer Ordnung: Java

- Java: keine direkte Syntax für Funktionen höherer Ordnung
- Folgendes ist nicht möglich:

```
interface Collection {
  Object fold(Object f(Object a, Collection c), Object a); }
```

- Aber folgendes:

```
interface Foldable { Object f (Object a); }

interface Collection { Object fold(Foldable f, Object a); }
```

- Vergleiche Iterator aus Collections Framework (Java SE 6):

```
public interface Iterator<E> {
  boolean hasNext();
  E next(); }
```

- Seit Java SE 8 (März 2014): Anonyme Funktionen (Lambda-Ausdrücke)

Funktionen Höherer Ordnung: C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
extern list filter(int f(void *x), list l);
```

```
extern list map(void *f(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Benutzung: qsort (C-Standard 7.20.5.2)

```
include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Funktionen Höherer Ordnung: C

Implementierung von map:

```
list map(void *f(void *x), list l)
{
  list c;
  for (c= l; c!= NULL; c= c-> next) {
    c-> elem= f(c-> elem);
    }
  return l;
}
```

- Typsystem zu schwach:

```
  {
    *(int *)x= *(int *)x*2;
    return x;
  }
  void prt(void *x)
```

```
  printf("List 3: "); mapM_(prt, l); printf("\n");
```

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:
  - Einfache Rekursion entspricht foldr