



Funktionale Programmierung

Woche 8 – Fold, List Comprehensions, Kalender Beispiel

René Thiemann

Philipp Dablander

Joshua Ocker

Michael Schaper

Lilly Schönherr

Adam Pescoller

Letzte Vorlesung

- Funktionen höherer Ordnung
 - Funktionen sind Werte
 - Funktionen können Funktionen als Eingabe haben oder Funktionen als Ausgabe liefern
- Partielle Anwendung: für $f :: a \rightarrow b \rightarrow c \rightarrow d$ sind folgende Ausdrücke möglich

$f :: a \rightarrow b \rightarrow c \rightarrow d$

$f \text{ expr} :: b \rightarrow c \rightarrow d$

$f \text{ expr expr} :: c \rightarrow d$

- Sektionen: $(x >)$ und $(> x)$
- λ -Abstraktionen: $\lambda \text{ pat1} \dots \text{patN} \rightarrow \text{expr}$
- η -Kontraktion: $f \text{ pat1} \dots \text{patN} x = \text{expr } x$ kürzen zu $f \text{ pat1} \dots \text{patN} = \text{expr}$
- Beispiele von Funktionen höherer Ordnung
 - $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 - $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
 - $\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

Fold-Funktionen für Listen

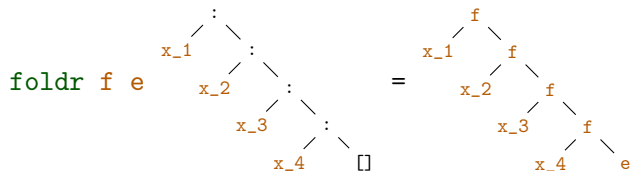
Die foldr Funktion

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

`foldr f e [] = e`

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

- `foldr f e` bietet strukturelle Rekursion auf Listen
 - `e` ist das Ergebnis des Basisfalls
 - `f` beschreibt, wie das Ergebnis anhand des ersten Listenelements und des rekursiven Ergebnisses berechnet wird
- `foldr f e` ersetzt `:` durch `f` und `[]` durch `e`



`foldr f e [x1, x2, x3, x4] = x1 `f` (x2 `f` (x3 `f` (x4 `f` e)))`

Ausdrucksstärke von `foldr`

- `foldr f e` ersetzt : durch `f` und `[]` durch `e`;
`foldr f e [x_1, x_2, x_3, x_4] = x_1 `f` (x_2 `f` (x_3 `f` (x_4 `f` e)))`
- `foldr f e` bietet strukturelle Rekursion auf Listen
- Konsequenz: **alle** Funktionsdefinitionen, die strukturelle Rekursion auf Listen verwenden, können über `foldr` definiert werden
- Beispiel-Definitionen über `foldr`
`sum = foldr (+) 0`
`product = foldr (*) 1`
`concat = foldr (++) [] -- merge list of lists into one list`
`xs ++ ys = foldr (:) ys xs`
`length = foldr (\ _ -> (+ 1)) 0`
`map f = foldr ((:) . f) []`
`all f = foldr ((&&) . f) True -- do all elements satisfy predicate?`

Varianten von foldr

-- foldr from previous slide

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

```
foldr f e [x_1, x_2, x_3] = x_1 `f` (x_2 `f` (x_3 `f` e))
```

-- foldr without starting element, only for non-empty lists

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

```
foldr1 f [x_1, x_2, x_3] = x_1 `f` (x_2 `f` x_3)
```

-- application: maximum of list elements

```
maximum = foldr1 max
```

-- foldl, apply function starting from the left

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

```
foldl f e [x_1, x_2, x_3] = ((e `f` x_1) `f` x_2) `f` x_3
```

-- application: reverse

```
reverse = foldl (flip (:)) []
```

Weitere Prelude-Funktionen für Listen

Take-While, Drop-While

- `takeWhile :: (a -> Bool) -> [a] -> [a]` und
`dropWhile :: (a -> Bool) -> [a] -> [a]`
 - `takeWhile p xs` nimmt Elemente von links von `xs`, solange `p` erfüllt ist
 - `dropWhile p xs` entfernt Elemente von links von `xs`, solange `p` erfüllt ist
 - Gleichheit: `takeWhile p xs ++ dropWhile p xs = xs`
- Kombinationen – effizientere Versionen der folgenden Definitionen
 - `splitAt :: Int -> [a] -> ([a], [a])`
`splitAt n xs = (take n xs, drop n xs)`
 - `span :: (a -> Bool) -> [a] -> ([a], [a])`
`span p xs = (takeWhile p xs, dropWhile p xs)`

Beispiel Anwendung: Separiere Wörter

- Aufgabe: Schreibe eine Funktion `words :: String -> [String]`, die eine Zeichenkette in Wörter zerlegt
- Beispiel: `words "I am fine. " = ["I", "am", "fine."]`

- Implementierung:

```
words s = case dropWhile (== ' ') s of
  "" -> []
  s1 -> let (w, s2) = span (/= ' ') s1
        in w : words s2
```

- Anmerkungen
 - nicht-triviale Rekursion auf Listen
 - `words` ist bereits vordefiniert
 - `unwords :: [String] -> String` ist die Umkehrfunktion und fügt Leerzeichen ein
 - ähnliche Funktionen zum Aufteilen an Zeilenumbrüchen oder zum Einfügen von Zeilenumbrüchen

```
lines :: String -> [String]
unlines :: [String] -> String
```

Elementweise Verknüpfung Zweier Listen

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
`zipWith f [x1, ..., xm] [y1, ..., yn] = [x1 `f` y1, ..., xmin{m,n} `f` ymin{m,n}]`
 - die resultierende Liste hat die Länge der kürzeren Eingabe-Liste
 - obige Gleichung ist kein Haskell-Code, denken Sie selbst über die rekursive Definition nach
- Spezialisierung `zip`
`-- (,) :: a -> b -> (a, b) is the pair constructor`
`zip :: [a] -> [b] -> [(a, b)]`
`zip = zipWith (,)`
- Umkehrfunktion: `unzip :: [(a, b)] -> ([a], [b])`
- Beispiele
 - `zip [1, 2, 3] "ab" = [(1, 'a'), (2, 'b')]`
 - `unzip [(1, 'c'), (2, 'b'), (3, 'a')] = ([1, 2, 3], "cba")`
 - `zipWith (*) [1, 2] [3, 4, 5] = [1*3, 2*4] = [3, 8]`

Anwendung: Prüfung, ob eine Liste sortiert ist

```
isSorted :: Ord a => [a] -> Bool
```

```
isSorted xs = all id $ zipWith (<=) xs (tail xs)
```

- `id :: a -> a` ist die Identitäts-Funktion `id x = x`;
die als “Prädikat” genutzt wird, ob ein Boolean wahr ist
- `($)` ist der Anwendungs-Operator mit geringer Präzedenz, `f $ x = f x`,
wird genutzt, um Klammern zu vermeiden
- Beispiel:

```
isSorted [1, 2, 5, 3]
= all id $ zipWith (<=) [1, 2, 5, 3] [2, 5, 3]
= all id [1 <= 2, 2 <= 5, 5 <= 3]
= all id [True, True, False]
= id True && id True && id False && True
= False
```

Operator-Tabelle mit Präzedenzen

Präzedenz	Operator	Assoziativität
9	!!, .	links(!!), rechts(.)
8	^, ^^, **	rechts
7	*, /, `div`	rechts
6	+, -	links
5	:, ++	rechts
4	==, /=, <, <=, >, >=	keine
3	&&	rechts
2		rechts
1	>>, >>=	links
0	\$	rechts

- alle Operatoren $\wedge$, $\wedge\wedge$, $**$ implementieren Potenzierung;
Unterschied liegt bei erlaubten Exponenten ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$) und Typ der Basis
- Operatoren ($>>$) und ($>>=$) werden später erläutert

List Comprehension

List Comprehension

- **List Comprehension** ist vergleichbar mit Mengen-Schreibweise in der Mathematik
- präzise, lesbare Definition
 - Summe der gerade Quadratzahlen bis 100: $\sum\{x^2 \mid x \in \{0, \dots, 100\}, \text{even}(x)\}$
- Beispiele von List Comprehensions in Haskell

```
evenSquares100 = sum [ x^2 | x <- [0 .. 100], even x]
```

```
prime n = n >= 2 && null [ x | x <- [2 .. n - 1], n `mod` x == 0]
```

```
pairs n = [ (i, j) | i <- [0..n], even i, j <- [0..i]]
```

```
> pairs 5
```

```
[(0,0),(2,0),(2,1),(2,2),(4,0),(4,1),(4,2),(4,3),(4,4)]
```

List Comprehension – Struktur

```
foo zs = [ x + y + z |  
  x <- [0..20],  
  even x,  
  let y = x * x,  
  y < 200,  
  Just z <- zs]
```

- eine List Comprehension hat die Form `[e | Q]` wobei
 - `e` ein Haskell Ausdruck ist, z.B., `x + y + z`, und
 - `Q` ein **Qualifier** ist, d.h., eine Komma-separierte Liste von:
 - **Generatoren** der Form `pat <- expr`, wobei `expr` einen Listen-Typen hat, z.B., `x <- [0..20]` oder `Just z <- zs`;
`e` und spätere Teile des Qualifiers können Variablen in `pat` nutzen
 - **Guards**, d.h., Boolesche Ausdrücke, z.B., `even x` oder `y < 200`
 - **lokale Definitionen** der Form `let defs` (kein `in!`);
`e` und spätere Teile des Qualifiers können Variables und Funktionen nutzen, die in `defs` definiert wurden

wenn `Q` leer ist, schreibt man einfach `[e]`

List Comprehension – Übersetzung

```
[ x + y | x <- [0..20], even x, let y = x * x, y < 200]
```

- List Comprehension hat Form `[e | Q]`, wobei `Q` eine Liste aus Guards, Generatoren und lokalen Definitionen ist
- Semantik: List Comprehensions werden mit Hilfe von `concatMap` übersetzt

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

```
concatMap f = concat . map f
```

- Guards:

```
[e | b, Q] = if b then [e | Q] else []
```

- lokale Definitionen:

```
[e | let defs, Q] = let defs in [e | Q]
```

- Generatoren für vollständige Pattern (z.B., Variable oder Tupel von Variablen):

```
[e | pat <- xs, Q] = concatMap (\ pat -> [e | Q]) xs
```

- Generatoren für den allgemeinen Fall:

```
[e | pat <- xs, Q] = concatMap  
  (\ x -> case x of { pat -> [e | Q]; _ -> [] } )  
  xs      -- where x must be a fresh variable name
```


List Comprehension – Beispiel Übersetzungen

- Übersetzung

```
[e | b, Q] = if b then [e | Q] else []
```

```
[e | let defs, Q] = let defs in [e | Q]
```

```
[e | pat <- xs, Q] = concatMap (\ pat -> [e | Q]) xs
```

- Beispiele

```
[s | (s, g) <- xs, g == 1]
```

```
= concatMap ( \ (s, g) -> [s | g == 1]) xs
```

```
= concatMap ( \ (s, g) -> if g == 1 then [s] else []) xs
```

```
[y + z | x <- xs, let y = x * x, z <- [0 .. y]]
```

```
= concatMap ( \ x -> [y + z | let y = x * x, z <- [0 .. y]] ) xs
```

```
= concatMap ( \ x -> let y = x * x in [y + z | z <- [0 .. y]] ) xs
```

```
= concatMap ( \ x -> let y = x * x in  
    concatMap ( \ z -> [y + z] ) [0 .. y] ) xs
```

Beispiel Anwendung – Pythagoreische Tripel

- (x, y, z) ist **Pythagoreisches Tripel** gdw. $x^2 + y^2 = z^2$
- Aufgabe: finde alle Pythagoreischen Tripel in gegebenen Zahlenbereich

```
ptriple x y z = x^2 + y^2 == z^2
```

```
ptriples n = [ (x,y,z) |  
  x <- [1..n], y <- [1..n], z <- [1..n], ptriple x y z]
```

- es gibt Probleme mit Duplikaten wegen der Symmetrie

```
> ptriples 5  
[(3,4,5), (4,3,5)]
```

- folgende Lösung eliminiert Symmetrien, und ist auch effizienter

```
ptriples n = [ (x,y,z) |  
  x <- [1..n], y <- [x..n], z <- [y..n], ptriple x y z]
```

```
> ptriples 5  
[(3,4,5)]
```

Anwendung – Erstellung eines Monats-Kalenders

Erstellung eines Kalenders

- Aufgabe: gegeben ein Monat und ein Jahr, erstelle einen Monats-Kalender
- Beispiel: November 2025

Mo	Tu	We	Th	Fr	Sa	Su
					1	2
3	4	5	6	7	8	9
...						

- Dekomposition liefert zwei Teil-Aufgaben
 - Berechnungs-Phase (Monatsanfang, Schaltjahre, ...)
 - Layout und Darstellung
- hier: Fokus auf Layout und Darstellung, Berechnungs-Phase wird bereitgestellt

```
type Month    = Int
type Year      = Int
type Dayname  = Int -- Mo = 0, Tu = 1, ..., So = 6
-- monthInfo returns name of 1st day in m. and number of days in m.
monthInfo :: Month -> Year -> (Dayname, Int)
```

Der Picture Typ

- stelle Kalender als Bild dar, d.h., Liste von Zeilen, wobei jede Zeile eine Liste von Zeichen ist

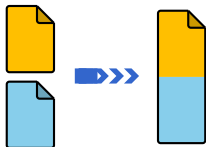
- Repräsentation in Haskell

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- betrachte Bild (h , w , rs)
- $rs :: [[Char]]$ – “Liste von Zeilen”
- Invariante 1: Länge von rs ist genau die Höhe h
- Invariante 2: alle Zeilen (also die Elemente von rs) haben Länge w
- Erzeugung eines Bildes aus einer einzigen Zeile

```
row :: String -> Picture
row r = (1, length r, [r])
```

Stapeln von Bildern



Stapeln zweier Bilder

```
above :: Picture -> Picture -> Picture
(h, w, css) `above` (h', w', css')
  | w == w'    = (h + h', w, css ++ css')
  | otherwise = error "above: different widths"
```

Stapeln mehrere Bilder

```
stack :: [Picture] -> Picture
stack = foldr1 above
```

Verschmelzen von nebeneinander stehenden Bildern



Verschmelzen von zwei benachbarten Bildern

```
beside :: Picture -> Picture -> Picture
(h, w, css) `beside` (h', w', css')
  | h == h'    = (h, w + w', zipWith (++) css css')
  | otherwise = error "beside: different heights"
```

Verschmelzen mehrerer benachbarter Bilder

```
spread :: [Picture] -> Picture
spread = foldr1 beside
```

Kombination von Stapeln und Verschmelzen

```
tile :: [[Picture]] -> Picture -- [[pic1,pic2,pic3],    ->  pic1pic2pic3
tile = stack . map spread      -- [pic4,pic5,pic6]]    ->  pic4pic5pic6
```

Erstellung von vielen Tages-Bildern

- wie erwähnt, nehmen wir an, dass folgende Funktion existiert

```
monthInfo :: Month -> Year -> (Dayname, Int) -- (first day, nr of days)
-- daynames are 0 (Monday), 1 (Tuesday), ...
```

```
daysOfMonth :: Month -> Year -> [Picture] -- 42 small pictures of size 1*3
```

```
daysOfMonth m y =
```

```
  map (row . rjustify 3 . pic) [1 - d .. numSlots - d]
```

```
  where
```

```
    (d, t) = monthInfo m y
```

```
    numSlots = 6 * 7 -- max 6 weeks * 7 days per week
```

```
    pic n = if 1 <= n && n <= t then show n else ""
```

```
rjustify :: Int -> String -> String
```

```
rjustify n xs
```

```
  | l <= n = replicate (n - l) ' ' ++ xs
```

```
  | otherwise = error ("text (" ++ xs ++ ") too long")
```

```
  where l = length xs
```


Zusammenbau des Gesamt-Bildes

- `daysOfMonth` liefert 42 Bilder (der Größe 1×3)
- benötigt: Layout + Kopfzeile für das Gesamt-Bild (der Größe 7×21)

```
month :: Month -> Year -> Picture
```

```
month m y = above weekdays . tile . groupsOfSize 7 $ daysOfMonth m y
  where weekdays = row " Mo Tu We Th Fr Sa Su"
```

```
-- groupsOfSize splits list into sublists of given length
```

```
groupsOfSize :: Int -> [a] -> [[a]]
```

```
groupsOfSize n [] = []
```

```
groupsOfSize n xs = ys : groupsOfSize n zs
```

```
  where (ys, zs) = splitAt n xs
```

Ausgabe eines Monats

- transformiere `Picture` in `String`
`showPic :: Picture -> String`
`showPic (_, _, css) = unlines css`
- liefere Resultat von `month m y` als `String`
`showMonth :: Month -> Year -> String`
`showMonth m y = showPic $ month m y`
- Anzeige des Strings mittels `putStr :: String -> IO ()`,
damit Zeilenumbrüche als solche ausgegeben werden

```
> showMonth 11 2025
```

```
" Mo Tu We Th Fr Sa Su\n                                1  2\n                                3  ..."
```

```
> putStr $ showMonth 11 2025
```

```
Mo Tu We Th Fr Sa Su
                1  2
  3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Zusammenfassung

- vielseitige Funktion für Listen: `foldr`, `foldl`, `foldr1`

- weitere nützliche Funktionen

<code>take</code> , <code>drop</code> , <code>splitAt</code> ,	-- split list at position
<code>takeWhile</code> , <code>dropWhile</code> , <code>span</code> ,	-- split list via predicate
<code>zipWith</code> , <code>zip</code> , <code>unzip</code> ,	-- (un)zip two lists
<code>concatMap</code> ,	-- map with concat combined
<code>($\\$)</code>	-- application operator

- Tabelle der Operatoren mit Präzedenzen

- List Comprehension

- präzise Beschreibung von Listen, ähnlich zu Mengen-Schreibweise in der Mathematik
- wird in normale Ausdrücke mit Hilfe von `concatMap` übersetzt
- Beispiel:

```
[ (x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n], x^2 + y^2 == z^2 ]
```

- Anwendung: Monats-Kalender