



# Functional Programming

Week 8 – Fold, List Comprehension, Calendar Application

René Thiemann   Diana Gründlinger   Alexander Montag   Adam Pescoller

Department of Computer Science

## Last Lecture

- partial application: if  $f$  has type  $a \rightarrow b \rightarrow c \rightarrow d$ , then build expressions

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

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

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

- sections:  $(x >)$  and  $(> x)$
- $\lambda$ -abstractions:  $\backslash \text{pat} \rightarrow \text{expr}$
- higher-order functions
  - functions are values
  - functions can take functions as input or return functions as output
- example higher-order functions
  - $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
  - $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
  - $\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

# Fold-Functions on Lists

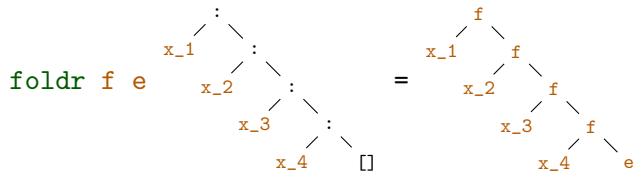
## The foldr Function

`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` captures structural recursion on lists
  - `e` is the result of the base case
  - `f` describes how to compute the result given the first list element and the recursive result
- `foldr f e` replaces `:` by `f` and `[]` by `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)))`

## Expressiveness of `foldr`

- `foldr f e` replaces `:` by `f` and `[]` by `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` captures structural recursion on lists
- consequence: **all** function definitions that use structural recursion on lists can be defined via `foldr`
- example definitions via `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?
```

## Variants of 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 (:)) []
```

# More Library Functions

## Take-While, Drop-While

- `takeWhile` :: (a -> Bool) -> [a] -> [a] and  
`dropWhile` :: (a -> Bool) -> [a] -> [a]
  - `takeWhile p xs` takes elements from left of `xs` while `p` is satisfied
  - `dropWhile p xs` drops elements from left of `xs` while `p` is satisfied
  - identity: `takeWhile p xs ++ dropWhile p xs = xs`
- combinations – more efficient versions of the following definitions
  - `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)`



## Example Application: Separate Words

- task: write function `words :: String -> [String]` that splits a string into words
- example: `words "I am fine. " = ["I", "am", "fine."]`

- implementation:

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

- notes

- non-trivial recursion on lists
- `words` is already predefined
- `unwords :: [String] -> String` is inverse which inserts blanks
- similar functions to split at linebreaks or to insert linebreaks

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

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

## Combining Two Lists

- `zipWith` :: (a -> b -> c) -> [a] -> [b] -> [c]  
`zipWith f [x1, ..., xm] [y1, ..., yn] = [x1 `f` y1, ..., xmin{m,n} `f` ymin{m,n}]`
- resulting list has length of shorter input
- above equality is not Haskell code, think about recursive definition yourself
- specialization `zip`  
-- (,) :: a -> b -> (a, b) is the pair constructor  
`zip` :: [a] -> [b] -> [(a, b)]  
`zip` = `zipWith` (,)
- inverse function: `unzip` :: [(a, b)] -> ([a], [b])
- examples
  - `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]`

## Application: Testing whether a List is Sorted

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

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

- `id :: a -> a` is the identify function `id x = x`;  
used as “predicate” whether a Boolean is `True`
- `($)` is application operator with low precedence, `f $ x = f x`,  
used to avoid parentheses
- example:

```
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
```

## Table of Precedences

precedence	operators	associativity
9	!!, .	left(!!), right(.)
8	^, ^^, **	right
7	*, /, `div`	left
6	+, -	left
5	:, ++	right
4	==, /=, <, <=, >, >=	none
3	&&	right
2		right
1	>>, >>=	left
0	\$	right

- all of ^, ^^, \*\* are for exponentiation: difference is range of exponents
- operators (>>) and (>>=) will be explained later

# List Comprehension

## List Comprehension

- **list comprehension** is similar to set comprehension in mathematics
- concise, readable definition
  - sum of even squares up to 100:  $\sum\{x^2 \mid x \in \{0, \dots, 100\}, \text{even}(x)\}$
- examples of list comprehension 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 – Structure

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

- list comprehension is of form `[e | Q]` where
  - `e` is Haskell expression, e.g., `x + y + z`
  - `Q` is the **qualifier**, a possibly empty comma-separated sequence of
    - **generators** of form `pat <- expr` where the expression has a list type, e.g., `x <- [0..20]` or `Just z <- zs`;  
`e` and later parts of qualifier may use variables of `pat`
    - **guards**, i.e., Boolean expressions, e.g., `even x` or `y < 200`
    - **local declarations** of form `let decls` (no `in!`);  
`e` and later parts of qualifier may use variables and functions introduced in `decls`
- if `Q` is empty, we just write `[e]`

## List Comprehension – Translation

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

- list comprehension is of form `[e | Q]` where qualifier is list of guards, generators and local definitions
- list comprehension is syntactic sugar, it is translated using the predefined function

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

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

- guards:

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

- local declaration:

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

- generators for exhaustive patterns (e.g., variable or pair of variables):

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

- generator (general case):

```
[e | pat <- xs, Q] = concatMap
```

```
(\ x -> case x of { pat -> [e | Q]; _ -> [] } )
```

```
xs      -- where x must be a fresh variable name
```



## List Comprehension – Translation Examples

- translations

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

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

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

- examples

```
[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
```

## Example Application – Pythagorean Triples

- $(x, y, z)$  is **Pythagorean triple** iff  $x^2 + y^2 = z^2$
- task: find all Pythagorean triples within given range

```
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]
```
- problem of duplicates because of symmetries

```
> ptriples 5
[(3,4,5), (4,3,5)]
```
- solution eliminates symmetries, also more efficient

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

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

## Application – Printing a Calendar

## Printing a Calendar

- given a month and a year, print the corresponding calendar

- example: November 2024

```
Mo Tu We Th Fr Sa Su
```

```
      1  2  3
```

```
 4  5  6  7  8  9 10
```

```
...
```

- decomposition identifies two parts
  - construction phase (computation of days, leap year, ...)
  - layout and printing
- we concentrate on printing, assuming machinery for construction

```
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)
```

## The Picture Type

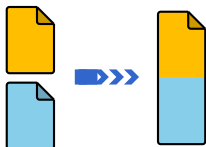
- encode calendar as a picture, i.e., a list of rows, where each row is a list of characters
- representation in Haskell

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

- consider  $(h, w, rs)$
- $rs :: [[Char]]$  – “list of rows”
- invariant 1: length of  $rs$  is height  $h$
- invariant 2: all rows (that is, lists in  $rs$ ) have length  $w$
- creation of a picture from a single row

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

## Stacking Pictures Above Each Other



## Stacking Two Picture Above Each Other

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

## Stacking Several Pictures Above Each Other

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

## Spreading Pictures Beside Each Other



## Spreading Two Pictures Beside Each Other

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

## Spreading Several Pictures Beside Each Other

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

## Tiling Several Pictures

```
tile :: [[Picture]] -> Picture
tile = stack . map spread
```

## Constructing a Month

- as indicated, assume function

```
monthInfo :: Month -> Year -> (Dayname, Int)
```

where daynames are 0 (Monday), 1 (Tuesday), ...

```
daysOfMonth :: Month -> Year -> [Picture]
```

```
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
```

```
  | 1 <= n = replicate (n - 1) ' ' ++ xs
```

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

```
  where l = length xs
```



## Tiling the Days

- `daysOfMonth` delivers list of 42 single pictures (of size  $1 \times 3$ )
- missing: layout + header for final picture (of size  $7 \times 21$ )

```
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
```

## Printing a Month

- transform a `Picture` into a `String`  
`showPic :: Picture -> String`  
`showPic (_, _, css) = unlines css`
- show result of `month m y` as `String`  
`showMonth :: Month -> Year -> String`  
`showMonth m y = showPic $ month m y`
- display final string via `putStr :: String -> IO ()` to properly print newlines and drop double quotes

```
> showMonth 11 2024
```

```
" Mo Tu We Th Fr Sa Su\n          1  2  3\n  4  5  6  7  8  9 10\n11 12 13 14 15 16 17\n18 19 20 21 22 23 24\n25 26 27 28 29 30"
```

```
> putStr $ showMonth 11 2024
```

```
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
```

## Summary

- versatile functions on lists: `foldr`, `foldl`, `foldr1`

- further useful functions on lists

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

- table of operator precedences

- list comprehension

- concise description of lists, similar to set comprehension in mathematics
- can automatically be translated into standard expressions based on `concatMap`
- example:

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

- calendar application