

# Functional Programming

## Higher-order functions

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2022/23

# Higher-order functions

- Functions are first-class citizens in Haskell
- A function can be
  - ▶ stored in data
  - ▶ argument of a (**higher-order**) functions
  - ▶ returned from a function

# Examples of higher-order functions

Most higher-order functions are polymorphic

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

```
filter  :: (a -> Bool) -> [a] -> [a]
```

# Examples of higher-order functions

Most higher-order functions are polymorphic

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

Example uses

```
> map even [1..5]
[False,True,False,True,False]
> filter even [1..10]
[2,4,6,8,10]
```

# Examples of higher-order functions

Most higher-order functions are polymorphic

```
map      :: (a -> b) -> [a] -> [b]
filter  :: (a -> Bool) -> [a] -> [a]
```

Example uses

```
> map even [1..5]
[False,True,False,True,False]
> filter even [1..10]
[2,4,6,8,10]
```

Haskell elides quantifiers in types

```
map  ::  $\forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]$ 
filter ::  $\forall a. (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$ 
```

# Instantiation

## Instances of a polymorphic type

Consider

$$\text{filter} \quad :: \quad \forall a. (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

We may replace (instantiate)  $a$  by any type:

$$\text{filter} \quad :: \quad (\text{Int} \rightarrow \text{Bool}) \rightarrow [\text{Int}] \rightarrow [\text{Int}]$$
$$\text{filter} \quad :: \quad ([\text{Char}] \rightarrow \text{Bool}) \rightarrow [[\text{Char}]] \rightarrow [[\text{Char}]]$$

# Instantiation

## Instances of a polymorphic type

Consider

$$\text{filter} \quad :: \quad \forall a. (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

We may replace (instantiate)  $a$  by any type:

$$\text{filter} \quad :: \quad (\text{Int} \rightarrow \text{Bool}) \rightarrow [\text{Int}] \rightarrow [\text{Int}]$$
$$\text{filter} \quad :: \quad ([\text{Char}] \rightarrow \text{Bool}) \rightarrow [[\text{Char}]] \rightarrow [[\text{Char}]]$$

## Instantiation rule

$$\frac{\text{INST} \quad e :: \forall a. t}{e :: t[a \mapsto t']}$$

# Function types

What's the difference between these types?

`Int -> Int -> Int`

`Int -> (Int -> Int)`

`(Int -> Int) -> Int`



# Function types

What's the difference between these types?

```
Int -> Int -> Int
```

```
Int -> (Int -> Int)
```

```
(Int -> Int) -> Int
```

How many arguments?

```
pick 1 = fst
```

```
pick 2 = snd
```

# Curried functions

## Compare these types

```
type T1 = Int -> Int -> Int
```

```
type T2 = (Int, Int) -> Int
```

- Both function types take two integers and return one
- T1 takes the arguments one at a time
- T2 takes both arguments as a pair

# Curried functions

## Compare these types

```
type T1 = Int -> Int -> Int
```

```
type T2 = (Int, Int) -> Int
```

- Both function types take two integers and return one
- T1 takes the arguments one at a time
- T2 takes both arguments as a pair

## Haskell prefers types like T1

- A **curried** type, after logician **Haskell B. Curry**
- Haskell's namesake
- Predefined functions `curry` and `uncurry` map between T1 and T2
- (an isomorphism)

# Typing Function Applications

## Typing Rule

$$\text{APP} \quad \frac{e_1 :: t_2 \rightarrow t_1 \quad e_2 :: t_2}{(e_1 \ e_2) :: t_1}$$

# Typing Function Applications

## Typing Rule

$$\text{APP} \quad \frac{e_1 :: t_2 \rightarrow t_1 \quad e_2 :: t_2}{(e_1 \ e_2) :: t_1}$$

## Example

Given

`map`  $:: \forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`even`  $:: \text{Int} \rightarrow \text{Bool}$

$$\frac{\frac{\text{map} :: \forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]}{\text{map} :: (\text{Int} \rightarrow \text{Bool}) \rightarrow [\text{Int}] \rightarrow [\text{Bool}]} \quad \text{even} :: \text{Int} \rightarrow \text{Bool}}{\text{map even} :: ??} \quad [1..5] :: [\text{Int}]$$
$$\text{map even } [1..5] :: ??$$

# Typing Function Applications

## Typing Rule

$$\text{APP} \quad \frac{e_1 :: t_2 \rightarrow t_1 \quad e_2 :: t_2}{(e_1 \ e_2) :: t_1}$$

## Example

Given

`map`  $:: \forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`even`  $:: \text{Int} \rightarrow \text{Bool}$

$$\frac{\frac{\text{map} :: \forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]}{\text{map} :: (\text{Int} \rightarrow \text{Bool}) \rightarrow [\text{Int}] \rightarrow [\text{Bool}]} \quad \text{even} :: \text{Int} \rightarrow \text{Bool}}{\frac{\text{map even} :: [\text{Int}] \rightarrow [\text{Bool}]}{\text{map even } [1..5] :: ??}} \quad [1..5] :: [\text{Int}]$$

# Typing Function Applications

## Typing Rule

$$\text{APP} \quad \frac{e_1 :: t_2 \rightarrow t_1 \quad e_2 :: t_2}{(e_1 \ e_2) :: t_1}$$

## Example

Given

`map`  $:: \forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`even`  $:: \text{Int} \rightarrow \text{Bool}$

$$\frac{\text{map} :: \forall a. \forall b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]}{\text{map} :: (\text{Int} \rightarrow \text{Bool}) \rightarrow [\text{Int}] \rightarrow [\text{Bool}] \quad \text{even} :: \text{Int} \rightarrow \text{Bool}} \quad \frac{\text{map even} :: [\text{Int}] \rightarrow [\text{Bool}] \quad [1..5] :: [\text{Int}]}{\text{map even } [1..5] :: [\text{Bool}]}$$

# Designing a higher-order function



# Designing a higher-order function

## Two functions on lists

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

```
product [] = 1
```

```
product (x:xs) = x * product xs
```

# Designing a higher-order function

## Two functions on lists

`sum [] = 0`

`sum (x:xs) = x + sum xs`

`product [] = 1`

`product (x:xs) = x * product xs`

## The common pattern

`g [] = z`

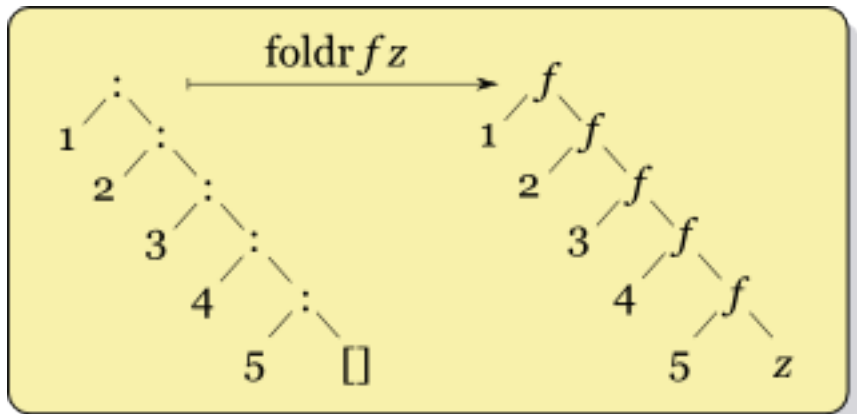
`g (x:xs) = x 'f' g xs`

with parameters

- `z :: b` is a value
- `f :: a -> b -> b` is a combining function

⇒ Represent `g` by some `foldr' f z`

## The foldr function: Intuition



# The foldr function

Making the pattern into a higher-order function

## Abstracting over value and combining function

```
foldr' f z [] = z
```

```
foldr' f z (x:xs) = x 'f' foldr' f z xs
```

where

- $z :: b$  is a value
- $f :: a \rightarrow b \rightarrow b$  is a combining function

# The foldr function

Making the pattern into a higher-order function

## Abstracting over value and combining function

```
foldr' f z []      = z
foldr' f z (x:xs) = x 'f' foldr' f z xs
```

where

- $z :: b$  is a value
- $f :: a \rightarrow b \rightarrow b$  is a combining function

What's the type of foldr'?

# The foldr function

Making the pattern into a higher-order function

## Abstracting over value and combining function

```
foldr' f z []      = z
foldr' f z (x:xs) = x 'f' foldr' f z xs
```

where

- $z :: b$  is a value
- $f :: a \rightarrow b \rightarrow b$  is a combining function

What's the type of foldr'?

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

# The foldr function

Making the pattern into a higher-order function

## Abstracting over value and combining function

```
foldr' f z []      = z
foldr' f z (x:xs) = x 'f' foldr' f z xs
```

where

- $z :: b$  is a value
- $f :: a \rightarrow b \rightarrow b$  is a combining function

What's the type of foldr'?

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

Also known as reduce

map + reduce = MapReduce

# Foldr in action

## sum and product

```
sum xs      = foldr (+) 0 xs
```

```
product xs = foldr (*) 1 xs
```



# Foldr in action

## sum and product

```
sum xs      = foldr (+) 0 xs
product xs = foldr (*) 1 xs
```

## more functions

```
or      xs = undefined
and     xs = undefined
concat  xs = undefined

maximum (x:xs) = undefined
```

# Foldr puzzles

f1

```
f1 xs = foldr (:) [] xs
```

# Foldr puzzles

f1

```
f1 xs = foldr (:) [] xs
```

f2

```
f2 xs ys = foldr (:) ys xs
```

# Foldr puzzles

f1

```
f1 xs = foldr (:) [] xs
```

f2

```
f2 xs ys = foldr (:) ys xs
```

f3

```
f3 xs = foldr snoc [] xs  
  where snoc x ys = ys++[x]
```

# Foldr puzzles

f1

```
f1 xs = foldr (:) [] xs
```

f2

```
f2 xs ys = foldr (:) ys xs
```

f3

```
f3 xs = foldr snoc [] xs  
  where snoc x ys = ys++[x]
```

f4

```
f4 f xs = foldr fc [] xs  
  where fc x ys = f x:ys
```

# Transforming functions

## Useful operations on functions

- partial application
- operator sections
- function composition
- anonymous functions aka lambda expressions
- eta conversion

# Partial application

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

```
take 5 :: [a] -> [a]
```

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

```
foldr (+) :: Int -> [Int] -> Int
```

```
foldr (+) 0 :: [Int] -> Int
```

- Partial application = function application with “too few” arguments
- Result is a function
- Can be used like any other function

# Operator sections

```
-- subtraction
(-) :: Int -> Int -> Int
-- subtract one
(- 1) :: Int -> Int
-- subtract from one
(1 -) :: Int -> Int

-- less than 0
(< 0) :: Int -> Bool
-- greater than 0
(0 >) :: Int -> Bool
```

can be done with every infix function



# Function composition

## Example

- Remove spaces from string as in this example

```
removeSpaces "abc def \n ghi" == "abcdefghi"
```

# Function composition

## Example

- Remove spaces from string as in this example

```
removeSpaces "abc def \n ghi" == "abcdefghi"
```

- In module `Data.Char`

```
isSpace :: Char -> Bool
```

# Function composition

## Example

- Remove spaces from string as in this example  
`removeSpaces "abc def \n ghi" == "abcdefghi"`
- In module `Data.Char`  
`isSpace :: Char -> Bool`
- yields definition  
`removeSpaces xs = filter (not . isSpace) xs`

# Function composition

## Example

- Remove spaces from string as in this example  
`removeSpaces "abc def \n ghi" == "abcdefghi"`
- In module `Data.Char`  
`isSpace :: Char -> Bool`
- yields definition  
`removeSpaces xs = filter (not . isSpace) xs`
- Operator `“.”` is function composition defined by  
$$(f \ . \ g) \ x = f \ (g \ x)$$

# Function composition

## Example

- Remove spaces from string as in this example  
`removeSpaces "abc def \n ghi" == "abcdefghi"`
- In module `Data.Char`  
`isSpace :: Char -> Bool`
- yields definition  
`removeSpaces xs = filter (not . isSpace) xs`
- Operator `“.”` is function composition defined by  
$$(f \ . \ g) \ x = f \ (g \ x)$$
- What's the type of `(.)`?

# Function composition

## Example

- Remove spaces from string as in this example  
`removeSpaces "abc def \n ghi" == "abcdefghi"`
- In module `Data.Char`  
`isSpace :: Char -> Bool`
- yields definition  
`removeSpaces xs = filter (not . isSpace) xs`
- Operator `“.”` is function composition defined by  
`(f . g) x = f (g x)`
- What's the type of `(.)`?
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

# Anonymous functions

- Usual function definition

`snoc x ys = ys++[x]`

# Anonymous functions

- Usual function definition

```
snoc x ys = ys++[x]
```

- Alternative: define `snoc` using an anonymous function aka **lambda expression**

```
snoc = \ x ys -> ys++[x]
```

Nowadays, the Unicode lambda can also be used



# Anonymous functions

- Usual function definition

```
snoc x ys = ys++[x]
```

- Alternative: define `snoc` using an anonymous function aka **lambda expression**

```
snoc = \ x ys -> ys++[x]
```

Nowadays, the Unicode `lambda` can also be used

- Often for function used in one place as in

```
f3 xs = foldr snoc [] xs  
  where snoc x ys = ys++[x]
```

# Anonymous functions

- Usual function definition

```
snoc x ys = ys++[x]
```

- Alternative: define `snoc` using an anonymous function aka **lambda expression**

```
snoc = \ x ys -> ys++[x]
```

Nowadays, the Unicode `lambda` can also be used

- Often for function used in one place as in

```
f3 xs = foldr snoc [] xs  
  where snoc x ys = ys++[x]
```

- Equivalently replace `snoc` by its definition

```
f3' xs = foldr (\ x ys -> ys++[x]) [] xs
```

# Anonymous functions

- Usual function definition

```
snoc x ys = ys++[x]
```

- Alternative: define snoc using an anonymous function aka **lambda expression**

```
snoc = \ x ys -> ys++[x]
```

Nowadays, the Unicode lambda can also be used

- Often for function used in one place as in

```
f3 xs = foldr snoc [] xs  
  where snoc x ys = ys++[x]
```

- Equivalently replace snoc by its definition

```
f3' xs = foldr (\ x ys -> ys++[x]) [] xs
```

- Shorten more using partial application

```
f3'' xs = foldr (\ x -> (++[x])) [] xs
```

# Eta conversion

- 1 A number of definitions have the form
$$f\ x = g\ x$$
where  $x$  does not occur in  $g$
- 2 In such cases, the formal parameter  $x$  is redundant:
$$f = g$$
is an equivalent definition.
- 3 The transformation from (1) to (2) is called **eta reduction**.<sup>1</sup>
- 4 The typing of an eta-reduced definition is more restricted.

---

<sup>1</sup>Reverse transformation: **eta expansion**; both directions: **eta conversion**

## Examples for eta-reduced definitions

```
sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
concat = foldr (++) []
removeSpaces = filter (not . isSpace)
```

# Exercises

```
takeLine :: String -> String
-- takeLine "abc\ndef\nghi\n" == "abc"

takeWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' :: (a -> Bool) -> [a] -> [a]
```

# Exercises

```
lines :: String -> [String]
-- lines "abc\ndef\nghi\n" == ["abc", "def", "ghi"]
```

```
segments' :: (a -> Bool) -> [a] -> [[a]]
```

```
words :: String -> [String]
-- words "abc def ghi" == ["abc","def","ghi"]
```

# Exercises

Define a function that counts how many times words occur in a text and displays each word with its count.

```
wordCounts :: String -> [String]
```

Example use

```
*Main> putStr (wordCounts "hello clouds\nhello sky")
clouds: 1
hello: 2
sky: 1
```



## Higher-order functions

- take functions as parameters,
- often have polymorphic types,
- abstract common patterns (map, filter, foldr),
- enable powerful programming techniques (partial application, operator sections, function composition, anonymous functions, eta conversion).