



# Functional Programming

## Lecture 2

Cezary Kaliszyk Jonas Schöpf  
Christian Sternagel Vincent van Oostrom

Department of Computer Science

## Overview

- Types and Type Classes
- Lists
- Patterns, Guards, and More
- Higher-Order Functions

## Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, **higher-order functions**, historical overview, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, **lists**, modules, **pattern matching**, **polymorphism**, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, **types and type classes**, unification, user-defined types

## Basic Concepts

- **types**  $\tau$  are built according to grammar

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid C \tau \dots \tau$$

- with **type variables**  $\alpha - a, b, \dots$
- **type constructors**  $C - \text{Bool}, \text{Int}, [], (), \dots$
- **function type constructor**  $\rightarrow$  (special case of previous item)
- $\rightarrow$  associates to the right:  $\tau \rightarrow (\tau \rightarrow \tau) = \tau \rightarrow \tau \rightarrow \tau$
- as approximation types may be thought of as collections of values their inhabitants can reduce to, e.g.,  $\text{Bool} = \{\text{True}, \text{False}\}$ , reflects the intuition that every expression of type  $\text{Bool}$  reduces either to  $\text{True}$  or  $\text{False}$  (or diverges) during runtime
- **type signature/constraint**  $e :: \tau$  means “ $e$  is of type  $\tau$ ”



## Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)
- `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)
- `Integer` – arbitrary-precision integers
- `Float` – single-precision floating-point numbers (`-12.34`, `3.14159`)
- `Double` – double-precision floating-point numbers

syntactic sugar for list of characters, e.g., `['a','b','c']`

## Note – Show Types in GHCi

- `Prelude> :set +t`
- commands may be put inside `~/ .ghci` (read on GHCi startup)

## Function Types

- type of functions from values of type  $\tau_1$  to values of type  $\tau_2$ :  $\tau_1 \rightarrow \tau_2$
- **every function** takes **single** argument and returns **single** result
- simulating multiple arguments: use tuples

## Examples

```
not :: Bool -> Bool
not True = False
not False = True
```

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

## List Types

- type of lists with elements of type  $\tau$ :  $[\tau]$
- all elements are of same type
- no restriction on length of list

## Tuple Types

- type of tuples with elements of types  $\tau_1, \dots, \tau_n$ :  $(\tau_1, \dots, \tau_n)$
- length: 2 (pair), 3 (triple), 4 (quadruple), ...,  $n$  ( $n$ -tuple), ...
- elements may be of different types
- fixed number of elements

## Examples

```
['a','b','c','d'] :: [Char]
["One","Two","Three"] :: [String]
[['a','b'],['c','d','e']] :: [[Char]]
(False,True) :: (Bool,Bool)
("Yes",True,'a') :: (String,Bool,Char)
```

## Currying

- transform function taking tuple as input into function returning another function as output
- in presence of **partial application**, curried functions are more versatile than uncurried functions

## Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

## Anonymous Functions – “Lambda-Abstractions”

- $\lambda x \rightarrow e$  is function taking  $x$  and returning  $e$

## Example

```
add' = \x -> \y -> x + y
```

## Basic Functions

- `Bool`: “conjunction” `&&`, “disjunction” `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`
- `(a, b)`: choose first `fst`, choose second `snd`

## Examples

```
not True      == False
(False && x)   == False
(True  || x)  == True
otherwise     == True
```

```
fst (x, y) == x
snd (x, y) == y
```

## The Eq Class – Equality

- specification, one of:
 

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

## The Ord Class – Orders

- prerequisite: `Eq`
- specification, one of:
 

```
compare :: Ord a => a -> a -> Ordering
(<=)    :: Ord a => a -> a -> Bool
```
- where `Ordering = {LT, EQ, GT}`
- additional functions: `(<)`, `(>=)`, `(>)`, `min`, `max`

## The Read Class – “from string”

- useful functions:
 

```
read :: Read a => String -> a
```

## Overloaded Types

- support standard set of operations
- use same name, independent of actual type

## Realization – Class Constrains

- syntax:  $e :: C\ a \Rightarrow \tau$
- meaning: “for every type `a` of class `C`, the type of `e` is  $\tau$ ” (where  $\tau$  does contain `a`)

## Example – Addition

(`op`) turns infix `op` into prefix

- $(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$
- “for every type `a` of class `Num`, addition has type `a -> a -> a`”
- since, e.g., `Int` is of class `Num`, we obtain that addition is of type `Int -> Int -> Int`, when used on `Ints`

## The Show Class – “to string”

- specification, one of:
 

```
show      :: Show a => a -> String
showsPrec :: Show a => Int -> a -> String -> String
```
- additional functions: `showList`

## The Num Class – Numeric Types

- prerequisites: `Eq` and `Show`
- specification, all of:
 

```
(+)      :: Num a => a -> a -> a
(*)      :: Num a => a -> a -> a
(-)      :: Num a => a -> a -> a
abs      :: Num a => a -> a
signum   :: Num a => a -> a
fromInteger :: Num a => Integer -> a
```
- additional functions: `negate`

visit: <http://haskell.org> → Documentation → Language Report: Haskell 2010

## Constructing Lists

- $[a] ::= [] \mid a : [a]$
- for given list, exactly two cases: either empty  $[]$ , or contains at least one element  $x$  and a remaining list  $xs$  ( $x : xs$ )
- $[x_1, x_2, \dots, x_n]$  abbreviates  $x_1 : (x_2 : (\dots : (x_n : []) \dots))$
- $(:)$  is right-associative, hence  $x_1 : (x_2 : xs) = x_1 : x_2 : xs$

## Examples

```
1 : (2 : (3 : (4 : []))) == 1 : 2 : 3 : 4 : []
1 : 2 : 3 : 4 : []      == [1,2,3,4]
1 : [2,3,4]             == [1,2,3,4]
```

## Testing for Emptiness

- $\text{null} :: [a] \rightarrow \text{Bool} - \text{True}$  iff argument is empty list

## Functions on Integer Lists

```
range m n =
  if m > n then []
  else m : range (m + 1) n
```

```
mySum xs =
  if null xs then 0
  else head xs + mySum (tail xs)
```

```
prod xs =
  if null xs then 1
  else head xs * prod (tail xs)
```

## Accessing List Elements – Selectors

- $\text{head} :: [a] \rightarrow a$  – extract first element (fail on empty list)
- $\text{tail} :: [a] \rightarrow [a]$  – drop first element (fail on empty list)

## A Polymorphic List Function

- **polymorphic** means “having many forms”
- definition
 

```
myReplicate n x =
  if n <= 0 then []
  else x : myReplicate (n - 1) x
```
- $\text{myReplicate}$  has type  $(\text{Ord } t, \text{Num } t) \Rightarrow t \rightarrow a \rightarrow [a]$
- can construct lists with elements of arbitrary type  $a$ , where length is given by some ordered numeric type  $t$

## Exercise

use equational reasoning to evaluate  $\text{myReplicate } 2 \text{ 'c'}$

## Examples

```
range 1 3 = [1,2,3]
range 3 2 = []
```

```
mySum [1,2,3] = 1 + 2 + 3 + 0
mySum [] = 0
```

```
prod [1,2,3] = 1 * 2 * 3 * 1
prod [] = 1
```

$$\text{mySum (range 1 n)} = \sum_{i=1}^n i$$

## Patterns

- used to match specific cases

- defined by

$\langle pat \rangle ::=$	<code>_</code>	wildcard
	<code>x</code>	variable pattern
	<code>x @ &lt;pat&gt;</code>	“as” pattern
	<code>[&lt;pat&gt;, ..., &lt;pat&gt;]</code>	list pattern
	<code>(&lt;pat&gt;, ..., &lt;pat&gt;)</code>	tuple pattern
	<code>C &lt;pat&gt; ... &lt;pat&gt;</code>	constructor pattern

- `_` matches everything and ignores the result
- `x` matches everything and binds the result to `x`
- `x @ <pat>` matches the same as `<pat>` and binds result to `x`
- constructor patterns match the described application of a data constructor (example constructors are `(:)` and `[]` for lists, `True` and `False` for Boolean values, ...)
- patterns may be used in arguments of function definitions and together with the `case` construct

## Pattern Guards

- a pattern may be followed by a guard `b`
- syntax: `<pat> | b`
- where `b` is a Boolean expression

## Example

```
f1 (x, _) | x >= 0 = x -- only if x non-negative
f2 (x:xs) | null xs = ... -- same as [x]
```

## The case Construct

```
case e of <pat1> -> e1
         :
         <patn> -> en
```

- checks `<pat1>` to `<patn>` top to bottom
- if `<pati>` is first match, `ei` is evaluated

## Example – Pattern Matching

```
mySum [] = ... -- constructor pattern
fst (x, _) = x -- patterns: tuple, variable, wildcard
case xs of [x] -> ... -- patterns: list, variable
          _ -> ... -- wildcard
```

## Refined Definitions

```
myReplicate n x | n <= 0 = []
                | otherwise = x : myReplicate (n - 1) x
```

```
range m n | m > n = []
           | otherwise = m : range (m + 1) n
```

```
mySum [] = 0
mySum (x:xs) = x + mySum xs
```

```
prod [] = 1
prod (x:xs) = x * prod xs
```

## Definition

a function is of **higher-order** if it takes functions as arguments

## Examples

```
twice f x = f (f x) -- apply f twice to x
```

## Sections

- abbreviation for partially applied infix operators
- `(x `op`)` abbreviates `(\y -> x `op` y)`
- `(`op` y)` abbreviates `(\x -> x `op` y)`

## Examples

```
ghci> twice (^2) 10
10000
```

## Processing Lists – filter

- possible definition
 

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs
```
- syntactic sugar `filter p xs = [x | x <- xs, p x]`

## Examples

```
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> filter (>5) [1..10]
[6,7,8,9,10]
ghci> filter (/= ' ') "abc def ghi"
"abcdefghi"
```

## Processing Lists – map

- possible definition
 

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```
- syntactic sugar `map f xs = [f x | x <- xs]`

## Examples

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]
ghci> import Data.Char
ghci> map isDigit ['a','1','b','2']
[False,True,False,True]
ghci> map reverse ["abc","def","ghi"]
["cba","fed","ihg"]
ghci> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

## “Fold Right” – A Very Expressive Function

- possible definition
 

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b [] = b
foldr f b (x:xs) = x `f` (foldr f b xs)
```
- `b` is 'base value'
- `f` combining function (binary)
- intuitively `foldr f b [x1,x2,...,xn]`

$$= \text{foldr } f \text{ b } (x_1 : (x_2 : \dots (x_n : []) \dots))$$

$$= (x_1 \text{ `f` } (x_2 \text{ `f` } \dots (x_n \text{ `f` } \text{ b } ) \dots))$$

## This Pattern is Very General

- take `(+)` for `f` and `0` for `b`: `foldr (+) 0 = sum`
- take `(*)` for `f` and `1` for `b`: `foldr (*) 1 = product`
- take `const (+1)` for `f` and `0` for `b`:  
`foldr (const (+1)) 0 = length` (where `const f _ = f`)

add dummy argument

## Homework (for November 9th)

1. Read Chapters 1 and 2 of [Real World Haskell](#).
2. Work through lessons 4 to 5 on <http://tryhaskell.org/>.
3. Give the types (and class constraints) for each of:

```
pair x y      = (x, y)
tail2 xs     = tail (tail xs)
triple x     = x * 3
thrice f x   = f (f (f x))
mapPair f (x, y) = (f x, f y)
idList      = filter (const True)
```

4. Use equational reasoning to stepwise compute the result of `filter (const False) ["a","b","c"]` on paper.
5. Using `foldr`, give alternative definitions of two of the functions we have seen so far (excluding those that we already defined via `foldr`).
6. Define a function `intercalate :: [a] -> [[a]] -> [a]` such that `intercalate xs xss` inserts the list `xs` between the lists in `xss` and concatenates the result.

### Example:

```
intercalate "; " ["one","two","six"] = "one; two; six"
```