



# Functional Programming

## Lecture 4

Cezary Kaliszyk   Jonas Schöpf  
Christian Sternagel   Vincent van Oostrom

Department of Computer Science

## Overview

- Intermediate Wrap-Up
- User-Defined Types
- Trees
- Input and Output

## 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

## Functions You Should Know

- infix operators and special syntax  
`(<=)`, `(<)`, `(==)`, `(>=)`, `(>)`, `(||)`, `(-)`, `(,)`, `(:)`, `(/=)`, `(.)`,  
`(*)`, `(&&)`, `(+)`, `(++)`, `[]`, `[m..n]`
- other Prelude functions  
`abs`, `compare`, `concat`, `const`, `div`, `drop`, `error`, `even`,  
`filter`, `foldr`, `foldr1`, `fromInteger`, `fst`, `head`, `init`, `last`,  
`length`, `lines`, `map`, `max`, `min`, `mod`, `negate`, `not`, `null`,  
`product`, `putStr`, `putStrLn`, `read`, `replicate`, `reverse`, `show`,  
`showList`, `showsPrec`, `signum`, `snd`, `splitAt`, `sum`, `tail`, `take`,  
`unlines`, `unwords`, `words`, `zip`, `zipWith`
- other Prelude constants  
`False`, `otherwise`, `True`
- other functions  
`Data.Char.isDigit`, `System.Environment.getArgs`

## Syntax You Should Recognize

- **anonymous functions** / functions without names  
`(\x -> 2 * x) -- an anonymous function for doubling`
- **infix operators and sections**
  - `(+)` = `(\x y -> x + y)`    infix to prefix
  - `x `f` y = f x y`    prefix to infix
  - `(a >)` = `(\x -> a > x)`    argument smaller than `a`?
  - `(> b)` = `(\x -> x > b)`    argument greater than `b`?
- **patterns and guards**  
`headIfPositive xs = case xs of`  
`x:_ | x > 0 -> x`
- **list comprehensions**  
`filter p xs == [x | x <- xs, p x]`  
`map f xs == [f x | x <- xs]`  
`concat (map f xs) == [y | x <- xs, y <- f x]`  
`concat $ map (\x -> map (,) x) ys ==`  
`[(x, y) | x <- xs, y <- ys]`

## Equational Reasoning

- a function definition in Haskell is a (set of conditional) equation(s)
- if conditions are met, we may “replace equals by equals”
- in this way we may **evaluate** function calls by applying equations stepwise, until we reach final result

## Kinds of Conditions

- “**if** `b` **then** `t` **else** `e`” is `t`, when `b` is true; and `e`, otherwise
- “**case** `e` **of** { `p1 -> e1`; ... ; `pn -> en` }” is `ei`, if `e` first matches `pi`

## Primitive Operations

- for primitive operations (like `(+)`, `(*)`, ...), we assume predefined equations
- e.g., `1 + 2 = 3`, `0 * 10 = 0`, ...

## Types and Type Classes

- **type signatures** – annotate functions by types  
`range :: Int -> Int -> [Int]`  
`range m n | m > n = []`  
`| otherwise = m : range (m + 1) n`
- **type synonyms** – mnemonic names for types  
`type Height = Int`  
`type Width = Int`
- **type classes** and **class constraints** – for every function `f`, specific to class `C`, type inference adds a `C`-constraint to type

## Example – Type Constraints

- without type signature, we get  
`ghci> :t range`  
`range :: (Ord a, Num a) => a -> a -> [a]`
- `m > n`, hence `m` and `n` of class `Ord` and `m` and `n` of same type
- `m + 1`, hence `m` of class `Num`
- `m` and `n` of same type, hence `n` of class `Num`

## Examples – Equational Reasoning

- definition  
`zip (x:xs) (y:ys) = (x, y) : zip xs ys`  
`zip _ _ = []`
- evaluate `zip [1,2,3] ['a','b']`
- definition  
`factorial n | n <= 1 = 1`  
`| otherwise = n * factorial (n - 1)`
- evaluate `factorial 3`
- definition `head xs = case xs of x:_ -> x`
- evaluate `head "ab"`
- definitions  
`null xs = case xs of { [] -> True; _ -> False }`  
`tail xs = case xs of _:ys -> ys`  
`prod xs = if null xs then 1`  
`else head xs * prod (tail xs)`
- evaluate `prod [5,6]`

## Data Declarations – Algebraic Data Types

- new types are introduced by

$$\begin{aligned} \mathbf{data} \ T \ \alpha_1 \ \cdots \ \alpha_n \ = \ & C_1 \ \tau_{11} \ \cdots \ \tau_{1m_1} \\ & | \ \vdots \\ & | \ C_k \ \tau_{k1} \ \cdots \ \tau_{km_k} \end{aligned}$$

- where  $T$  is name of new type (constructor)—starting with capital letter—taking  $n$  type parameters  $\alpha_1$  to  $\alpha_n$
- and  $C_i$  is name of  $i$ th (data) **constructor**, taking  $m_i$  arguments of types  $\tau_{i1}$  to  $\tau_{im_i}$  (with type variables among  $\alpha_1$  to  $\alpha_n$ )

### Examples

- `data Bool = False | True`
- `data List a = Nil | Cons a (List a)`
- `data Pair a b = Pair a b`

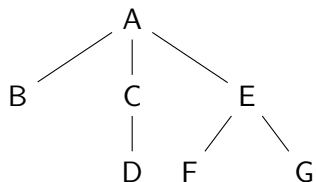
constructors and type names live in different name spaces

## Definition – Tree

- (rooted) tree  $T = (N, E)$
- with set of nodes/vertices  $N$
- and set of edges  $E \subseteq N \times N$
- unique root of  $T$  ( $root(T) \in N$ ) without predecessor
- all other nodes have exactly one predecessor

### Example

- $N = \{A, B, C, D, E, F, G\}$
- $E = \{(A, B), (A, C), (A, E), (C, D), (E, F), (E, G)\}$
- $root(T) = A$
- $T =$



## Automatically Deriving Type Class Instances

- for some type classes it is possible to automatically derive instances for algebraic data types
- e.g.,  
`data List a = Nil | Cons a (List a)`  
`deriving (Eq, Show, Read)`
- now, we are able to use `(==)`, `show`, and `read` for Lists

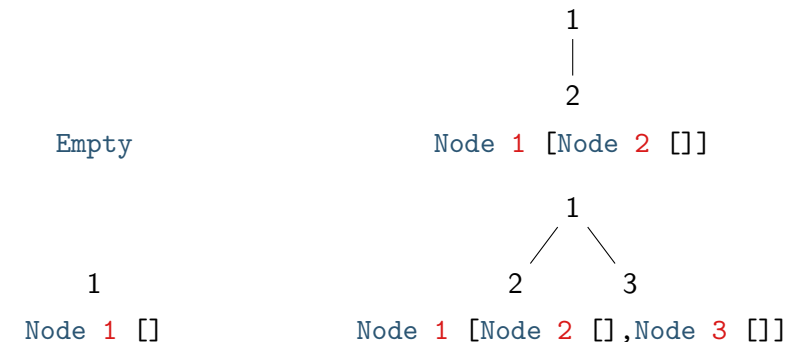
### Examples

```
ghci> Nil == Cons 1 Nil
False
ghci> show (Cons 1 (Cons 2 Nil))
"Cons 1 (Cons 2 Nil)"
ghci> read it :: List Int
Cons 1 (Cons 2 Nil)
```

## Trees in Haskell

- possible type for trees with arbitrary nodes  
`data Tree a = Empty | Node a [Tree a]`
- a tree is either empty (0 nodes) or there is at least one node with content of type `a` and an arbitrary number of successor trees

### Examples



## Binary Trees

- restrict number of successors (maximum 2)
- type
 

```
data BTree a = Empty | Node a (BTree a) (BTree a)
  deriving (Eq, Show, Read)
```

## Functions on Binary Trees

- size – number of nodes
 

```
size :: BTree a -> Integer
size Empty      = 0
size (Node _ l r) = size l + size r + 1
```
- height – length of longest path from root to some leaf
 

```
height :: BTree a -> Integer
height Empty    = 0
height (Node _ l r) = max (height l) (height r) + 1
```

## Transforming Trees into Lists

```
flatten Empty      = []
flatten (Node x l r) = flatten l ++ [x] ++ flatten r
```

## A Sorting Algorithm for Lists

```
sort = flatten . searchTree
```

## Creating Trees from Lists

- the easy way
 

```
fromList []      = Empty
fromList (x:xs) = Node x Empty (fromList xs)
```
- the balanced way
 

```
make [] = Empty
make xs = Node z (make ys) (make zs)
  where
    m          = length xs `div` 2
    (ys, z:zs) = splitAt m xs
```
- the orderly way
 

```
searchTree = foldr insert Empty
  where
    insert x Empty = Node x Empty Empty
    insert x (Node y l r)
      | x < y      = Node y (insert x l) r
      | otherwise = Node y l (insert x r)
```

## An Initial Example

- write the file `welcomeIO.hs`

```
main = do
  putStrLn "Greetings! What's your name?"
  name <- getLine
  putStrLn (
    "Welcome to Haskell's IO, " ++ name ++ "!")
```
- compile it with GHC via
 

```
$ ghc --make welcomeIO.hs
```
- and run it
 

```
$ ./welcomeIO
Greetings! What's your name?
```

## Notes

- `putStrLn` – prints string followed by newline
- `getLine` – reads line from standard input
- new syntax: `do` and `<-`

## IO and the Type System

- consider
 

```
ghci> :load welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine :: IO String
ghci> :t main
main :: IO ()
```
- `IO a` is type of IO actions delivering results of type `a` (in addition to their IO operations)

### Examples

- `String -> IO ()` – after supplying a string, we obtain an IO action (in case of `putStrLn`, “printing”)
- `IO ()` – just IO (in case of `main`, run our program)
- `IO String` – do some IO and deliver a string (in case of `getLine`, user-input)

## Using Pure Code Inside IO Actions

- consider program `reply.hs`

```
reply :: String -> String
reply name =
  "Pleased to meet you, " ++ name ++ ".\n" ++
  "Your name contains " ++ n ++ " characters."
  where
    n = show $ length name

main :: IO ()
main = do
  putStrLn "Greetings again. What's your name?"
  name <- getLine
  let niceReply = reply name
  putStrLn niceReply
```
- that is, we may use `let x = e` (there is no `in` here!) to bind result of pure expression `e` to name `x`

## Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification
- only `main` may start execution of IO actions
- inside IO actions, order is important; IO actions are executed in order of appearance (once execution starts); result of sequence of IO actions is result of **last** action
- inside IO actions, `x <- action` (where `action :: IO a`) may be used to bind result of `action` (which has type `a`) to name `x` (but seriously, this is actually only done, once execution starts)
- `x <- a` is not available outside IO actions

## Implications

- once we are inside an IO action, we cannot escape
- strict separation between purely functional code and IO
- when `IO a` does not appear inside type signature, we can be absolutely sure that no IO (“side-effect”) is performed

## Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

## Examples – Imitating Some GNU Commands

- `cat.hs` – print file contents
 

```
main = do
  [file] <- getArgs
  s <- readFile file
  putStr s
```
- `wc.hs` – count newlines/words/characters in input
 

```
count s = ns ++ " " ++ ws ++ " " ++ bs ++ "\n"
  where ns = show $ length $ lines s
        ws = show $ length $ words s
        bs = show $ length s

main = interact count
```
- `uniq.hs` – omit repeated lines of input
 

```
main = interact (unlines . nub . lines)
```
- `sort.hs` – sort input lines
 

```
main = interact (unlines . sort . lines)
```

## Homework (for November 23rd)

1. Read Chapter 7 of *Real World Haskell*.
2. Evaluate the function call `take 4 (iterate tail [1..3])` by equational reasoning using the definitions:
 

```
iterate f x = x : iterate f (f x)
tail (_:xs) = xs
take n xs | n <= 0 || null xs = []
take n (x:xs) = x : take (n-1) xs
```
3. Define a new type `Cmd` of commands that work with respect to an implicit stack and allow us to push an `Int` on top, pop the top element, and add the topmost elements by popping both and pushing the result on top. Moreover, implement a function
 

```
exec :: Cmd -> Stack Int -> Stack Int
```

 that executes a single command on a given stack.
4. Implement a function `levels :: BTree a -> [[a]]` that returns the list of nodes at each level of a binary tree.

### Example:

```
levels (Node 1 (Node 2 Empty Empty)
         (Node 3 Empty Empty)) = [[1],[2,3]]
```

## Notes

- `getArgs :: IO [String]` is in `System.Environment`
- `nub :: Eq a => [a] -> [a]` is in `Data.List`; eliminates duplicates
- `sort :: Ord a => [a] -> [a]` is in `Data.List`; sorts a list

## Do Some IO Action for Each Argument

- ```
foreach :: [a] -> (a -> IO ()) -> IO ()
```
- `foreach [] io = return ()`
  - `foreach (a:as) io = do { io a; foreach as io }`
- better `cat.hs`

```
main = do
  files <- getArgs
  if null files then interact id else do
    foreach files readAndPrint
  where readAndPrint file = do
        s <- readFile file
        putStr s
```

## Homework (for November 23rd, continued)

5. Implement a program `NL.hs` that echoes every line of its standard input with line numbers added.
 

```
Example: echo -e "what\nup?" | ./NL
1 what
2 up?
```
6. Implement a simple stack-based calculator that outputs the internal stack in each iteration, reads commands from standard input (until the string `exit` is observed) and executes them on the internal stack.
 

```
Example: ./Calc
stack: []
> Push 1
stack: [1]
> Push 2
stack: [2,1]
> Add
stack: [3]
> exit
```