



#### Functional Programming Lecture 10: Other Haskell Language Features

#### Viliam Lisý

Artificial Intelligence Center Department of Computer Science FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

## Example: Arithmetic Expressions

Recursive typed can represent tree structures, such as <u>expressions</u> from numbers, plus, multiplication.

data	Expr	-	Val	Int	
			Add	Expr	Expr
			Mul	Expr	Expr

1 + 2 \* 3

Add (Val 1) (Mul (Val 2) (Val 3))

Using recursion, it is now easy to define functions that process expressions. For example:

size :: Expr  $\rightarrow$  Int size (Val n) = 1size (Add x y) = size x + size ysize (Mul x y) = size x + size y eval :: Expr  $\rightarrow$  Int eval (Val n) = neval (Add x y) = eval x + eval y eval (Mul x y) = eval x \* eval y

## **Type Classes**

Functions required by a class can be accessed by

:info <classname>

:info Eq -- produces the following

Functions can often be implemented based on other only minimal complete definition is required (one of the above)

## Show Class

A class values convertible to a readable string

class Show a where
 showsPrec :: Int -> a -> ShowS
 show :: a -> String
 showList :: [a] -> ShowS

type ShowS = String -> String

This allows constant-time concatenation of results using function composition (optimization)

Minimal complete definition: showsPrec | show

#### Instance of a Class

A new instance can be added to a class by

instance Show Nat where
 show n = "N" ++ show (nat2int n)

#### **Class Contexts**

Remember the definition

To make Maybe an instance of Eq, a has to be in Eq

instance Eq a => Eq (Maybe a) where Nothing == Nothing = True (Just x) == (Just x') = x == x'

## **Deriving Classes**

Obvious definition of instances are automated

# **Defining Classes**

The implemented function bodies determine the minimum required functions

#### **Functor Class**

Class of structures you can map over

class Mapable f where mmap :: (a -> b) -> f a -> f b

instance Mapable[] where

mmap = map

instance Mapable Maybe where mmap f (Just x) = Just (f x) mmap f Nothing = Nothing

# Kinds

Types of types

- \* A specific type
- \* -> \* A type that given a type creates a type

:k

# **Types Summary**

- Everything has a type known in compile time
  - basic values
  - functions
  - data structures
- Types are key for data structures in Haskell
- Types can be instances of classes
   polymorphic functions
- "Types" of types are kinds

## **Higher Order Functions**

The same functions as in scheme are available

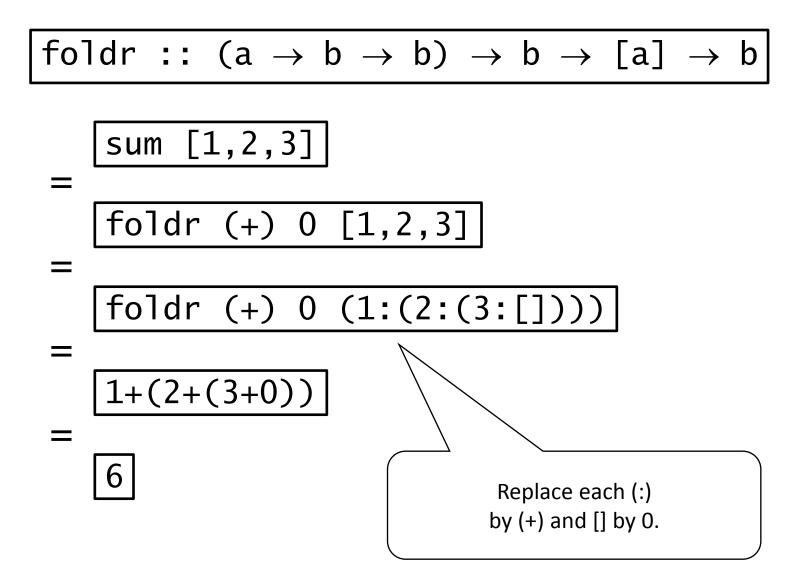
$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

filter :: (a  $\rightarrow$  Bool)  $\rightarrow$  [a]  $\rightarrow$  [a]

$$map f xs = [f x | x \leftarrow xs]$$

filter  $p xs = [x | x \leftarrow xs, p x]$ 

## Foldr



## Lambda Expressions

Functions can be constructed without naming the functions by using <u>lambda expressions</u>.

$$\lambda \mathbf{x} \rightarrow \mathbf{x} + \mathbf{x}$$

The symbol  $\lambda$  is typed as a backslash  $\setminus$ .

In mathematics, nameless functions are usually denoted using the  $\rightarrow$  symbol, as in  $x \rightarrow x + x$ .

As in scheme,

means

add = 
$$\lambda x \rightarrow (\lambda y \rightarrow x + y)$$

We also have the automated currying

add = 
$$\lambda x y \rightarrow x + y$$

We can use lambda expressions and local functions interchangeably

odds n = map f [0..n-1]  
where  
f x = 
$$x*2 + 1$$

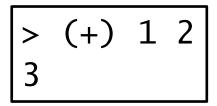
can be simplified to

odds n = map (
$$\lambda x \rightarrow x*2 + 1$$
) [0..n-1]

The earlier may be better if the local function has a natural name

#### **Operator Sections**

An <u>infix</u> operator can be converted into a curried <u>prefix</u> function by using parentheses.



This convention also allows one of the arguments of the operator to be included in the parentheses.

If  $\oplus$  is an operator then ( $\oplus$ ), (x $\oplus$ ) and ( $\oplus$ y) are called <u>sections</u>.

# **Infix Operators**

Any (prefix) function can become infix using ``
`mod`, `elem`

Names with only special symbols are infix

++++, +/+, %-

Precedence/asociativity of infix operators set by

prefix <0-9> <name>

Custom infix data constructors begin with :

:#, :+, :::

# **Infix Operators**

Information about associativity and precedence

:info

Interesting infix operators

. \$ unary -

# Modules

Haskell program is a collection of modules name spaces, abstract data declarations module names start with upper-cased character filenames must match module names in GHC

module <name> ( <exported>, <symbols> ) where

without exported symbols, everything is exported data constructors exported with type name Tree(Leaf,Branch), can be abbreviated to Tree(..)

#### Example Module

module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

# Importing Modules

Imports must be at the beginning of a module Prelude module is loaded by default We can choose names to import and hide

import Tree

import Tree hiding (tree1)

import Tree (tree1, fringe)

import qualified Tree as T hiding (tree1)



# Summary

- Type and type classes essential for Haskell
- Unnecessary, but pleasant Haskell features
  - higher order functions
  - lambda functions
  - infix operators and their sections
  - modules